THE UNIVERSITY OF NEW MEXICO

ECE 538
Advanced Computer Architecture

Instructor: Lei Yang

Department of Electrical and Computer Engineering

October 8, 2021

2021-10-06: End Class 13 here

- ✓ Instruction Set Architecture (MIPS ISA)
- ✓ Design Goals and Issues of ISA
- ✓ RISC V.S. CISC
- ✓ The Road to Future: RISC-V

Instruction Level Parallelism (ILP)

Referred to Chapter 3

&

(Appendix C in Book “Computer Organization and Design:

The Hardware/ Software Interface, Fifth Edition”)

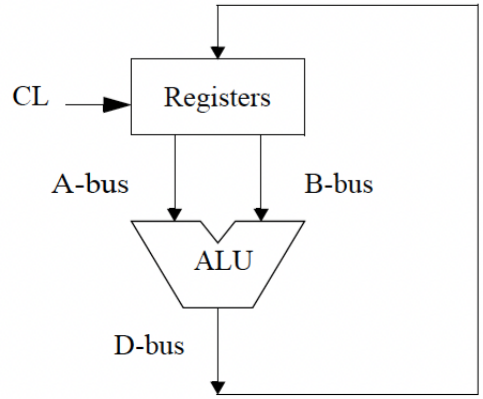
[Ref] Chapter Three in the textbook

How A CPU Works (Hardware + Software Parallelism)

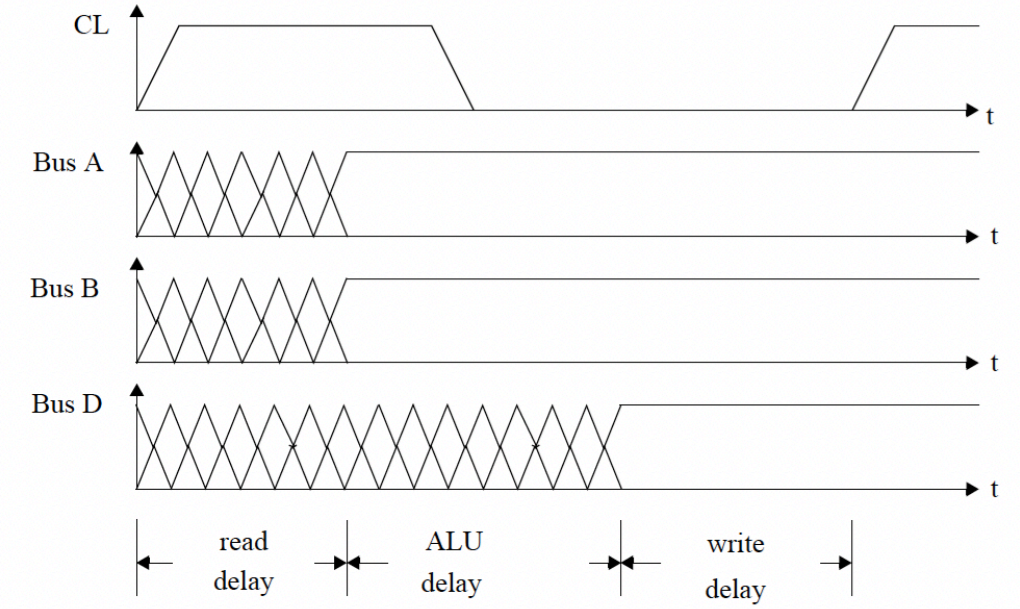
How A CPU Works (Hardware + Software Parallelism)

UNPIPELINED PROCESSOR

- Three-BUS Architecture:

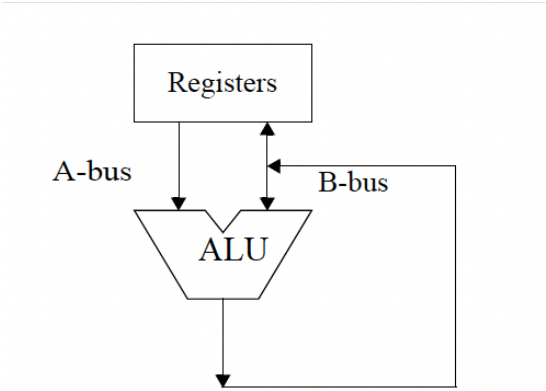


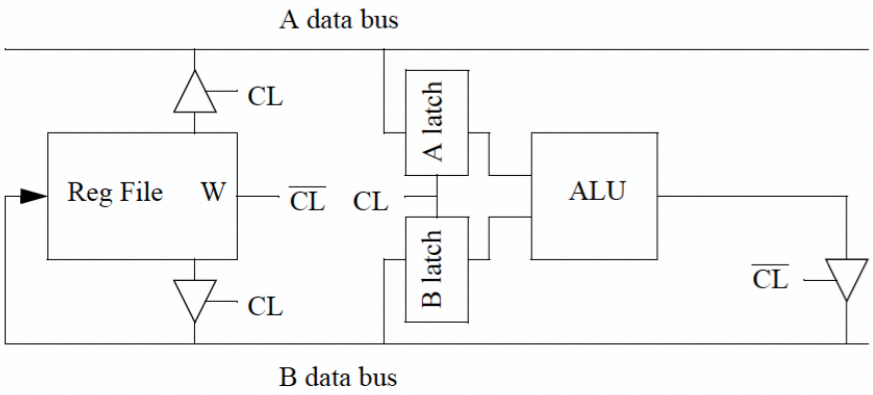
- Timing Diagram



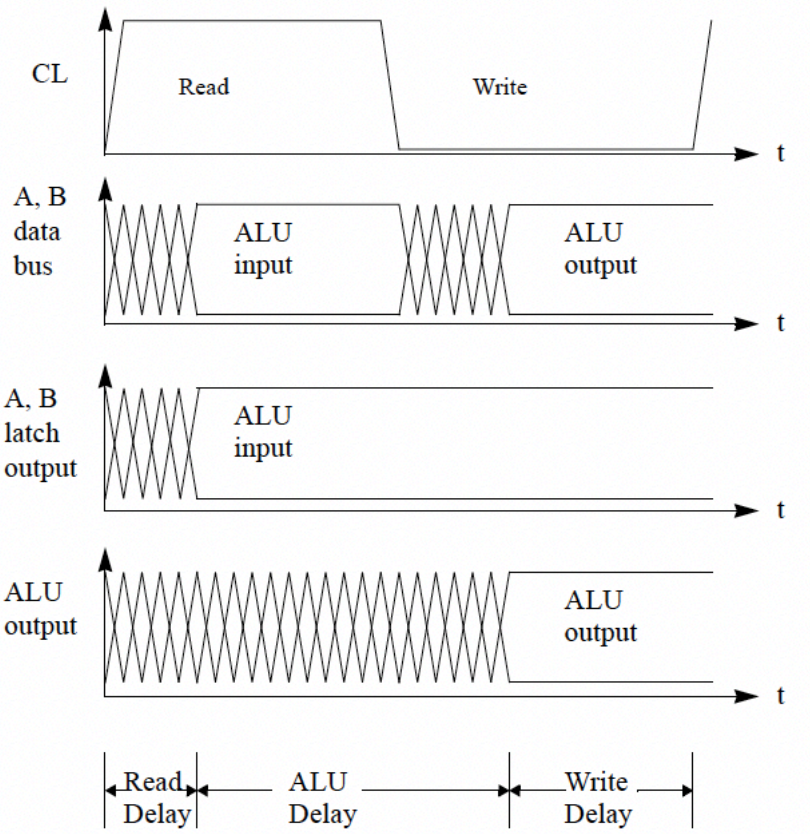
UNPIPELINED PROCESSOR

- Two-BUS Architecture:





- Timing Diagram

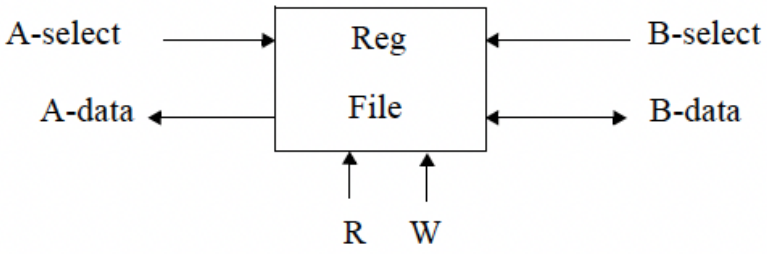


UNPIPELINED PROCESSOR

- For both 2-bus and 3-bus architectures:

Clock cycle time > Read Delay + ALU Delay + Write Delay

- The ALU is not doing anything useful during the read/write delays
- The register file is not doing anything during the ALU delay
- If there is no need to read and write into the register file at the same time, NO need to use 3-bus architecture



R	W	function
0	0	bbus not used by register file
0	1	write bbus into register selected by Bselect
1	0	read register selected by Bselect onto bbus
1	1	not allowed

Unpipelined Processor

- Performance Issues

  1. Longest delay determines clock period

  -- Critical path: load instruction

  -- Instruction memory -> Register file -> ALU -> Data memory -> Register file

  2. Not feasible to vary period for different instructions

  3. Violates design principle

  -- Making the common case fast

Improve performance by

Parallel Processing and *Pipelining…*

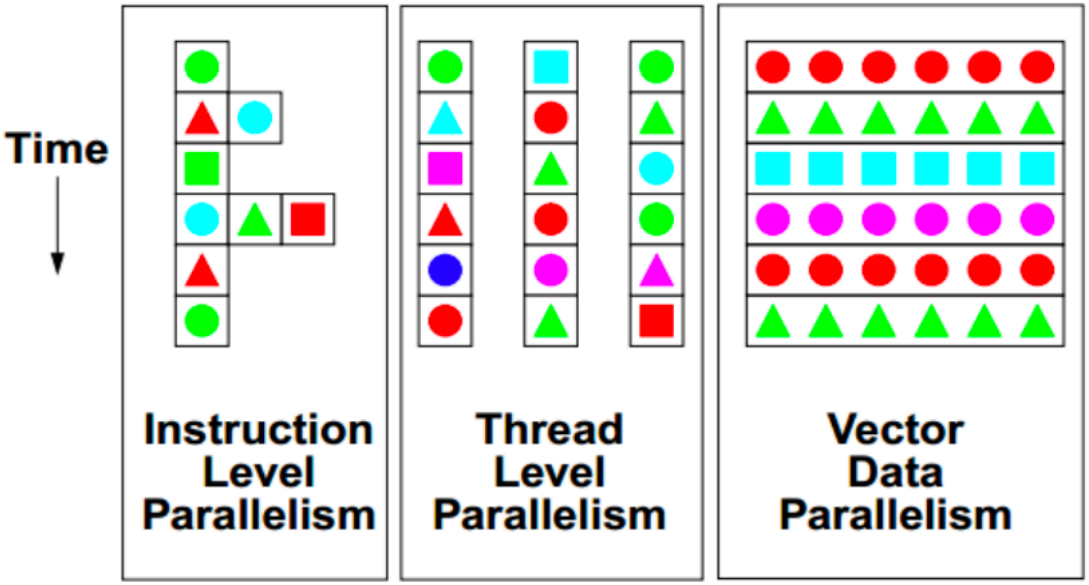
Basic concepts of
Parallel Processing and Pipelining

Types of Parallelism

- Instruction Level Parallelism (ILP)
  - Multiple instructions are executed concurrently
- Data Level Parallelism (DLP)
  - Each processor performs the same task on different data
- Task Level Parallelism (TLP)
  - Each processor performs different independent tasks

Overview – Three Levels of Parallelism

- ILP: Instruction Level Parallelism – e.g., Superscalar and VLIW processor
  - Out-of-Order execution (all in hardware)
  - Multiple independent instructions are identified and grouped to be executed concurrently in different functional units in a single processor
  - Can reduce CPI to values less than 1



Overview – Three Levels of Parallelism

- Other forms of parallelism
  - TLP: Thread/Task Level Parallelism – e.g., multicore and multi-processor systems
    - Multiprocessors, and HW Multithreading
    - Several independent threads/tasks are executed simultaneously
    - Reduce total execution time of multiple tasks
  - DLP: Data Level Parallelism – e.g., Vector processor and array processor
    - ✓ The same operation is performed on multiple data values concurrently in multiple processing units
    - ✓ Reduce instruction count to enhance performance

Serial and Parallel Computation

- Example-1: accountants adding the total amount of invoices

  Add the total amount in m=1024 invoices as quickly as possible by n accountants. Find the time (T) required to add these numbers?

The following assumptions can be used:

1. A stack of 1024 invoices, is initially given to accountant #1. The total amount of all the invoices are to be added.
2. Any accountant takes 1 second to add two numbers. (Counting time = 1 sec)
3. An accountant will require 3 seconds to send any number of invoices or computed results to any other accountant. (Transfer time = 3 sec)
4. Accountant #1 computes the final result.

Case-1: m=1024, number of accountants (n)=1:

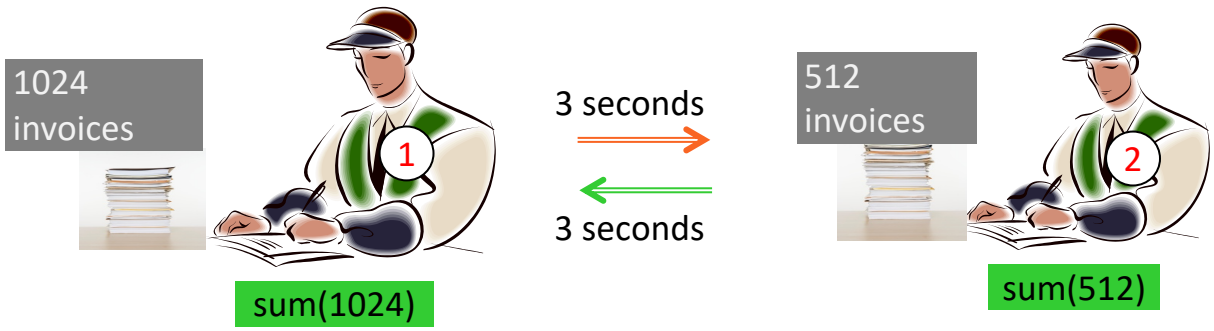
T=1023 seconds

only one accountant: this is an example of serial computation.

DISTRIBUTED AND PARALLEL COMPUTATION

Case-2:

m=1024, number of accountants (n)=2 (two accountants)



- 3 sec to transfer the invoices to Acc_2
- 511 sec for adding 512 invoices by Acc_1 and Acc_2
- 3 sec to transfer of partial results back to Acc_1
- 1 sec for sum (512) of Acc_1 and Acc_2
- 511×1 seconds to add 512 numbers

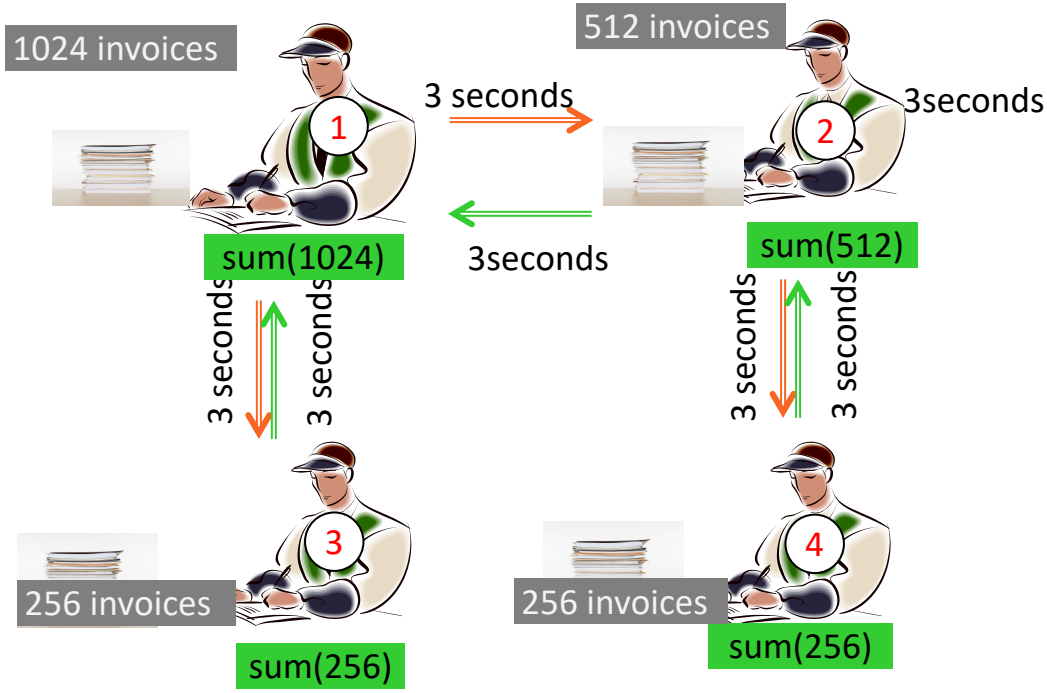
- Time to complete the task:

$$T = 3 \times 1 + 511 \times 1 + 3 \times 1 + 1 \times 1 = 3 + 511 + 3 + 1 = 518\ s$$

DISTRIBUTED AND PARALLEL COMPUTATION

Case-3:

m=1024, number of accountants (n)=4 (four accountants)

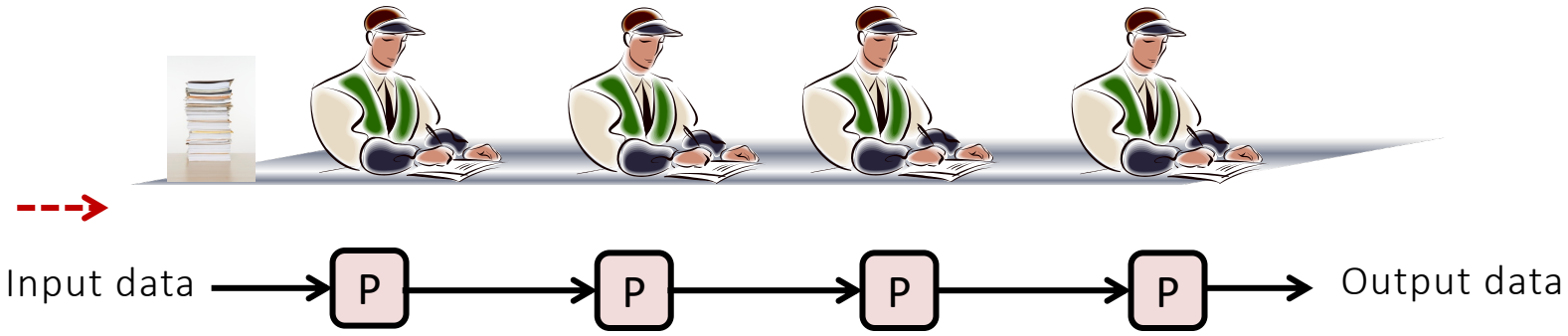


- 3 sec to transfer the invoices to Acc_2
- 3 sec to transfer the invoices to $Acc_3$and Acc_4
- 255 sec for adding 256 invoices by all Accountants
- 3 sec to transfer the partial results back to Acc_1 and Acc_2
- 1 sec for sum(256) by $Acc_1$and Acc_2
- 3 sec to transfer of partial results back to Acc_1
- 1 sec for sum(512) by Acc_1

- Time to complete the task:
- $T = 3\times2 + 255\times1 + 3\times2 + 1\times2 = 6 + 255 + 6 + 2 = 269\ s$

Derived concurrency - Pipelining

Concurrency within a task



- What is the advantage?

  -- Increased throughput.

- Any disadvantage?

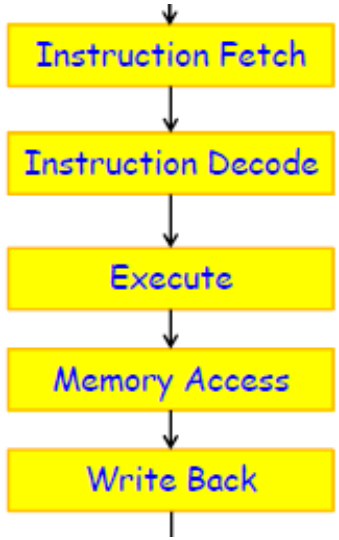
  -- Useful only when there is little/no intra-task dependency.

- What about latency?

  -- Increases with the number of pipeline stages.

Derived Concurrency - Pipelining

IF : instruction fetch
ID : instruction decode and data fetch
EX : execute operations or calculate address
MEM : memory operations
WB : write-back to register file



Clock cycle

1	2	3	4	5	6	7	8	9
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Data access corresponding to previous instruction can be performed while fetching the current instruction (in a same clock cycle)

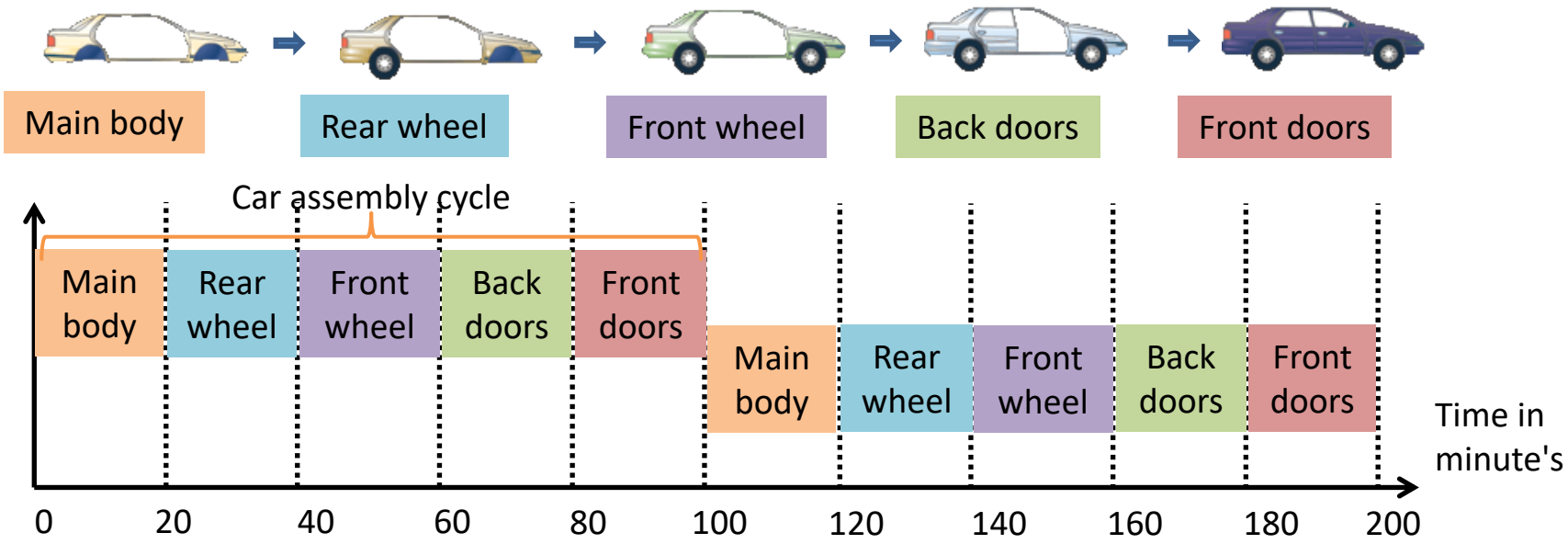
CPI (cycles per instruction): the value for a pipelined processor is the sum of the base CPI and all contributions from stalls

$$\boldsymbol{Pipeline\ CPI}$$

$$= Ideal\ pipeline\ CPI + Structural\ stalls + Data\ harzard\ stalls + Control\ stalls$$

Pipelining Analogy

- Pipelined: Building a car
- Non pipeline example – car assembly line
  - In each time unit, only one sub-task is active (other 4 sub-tasks are idle)



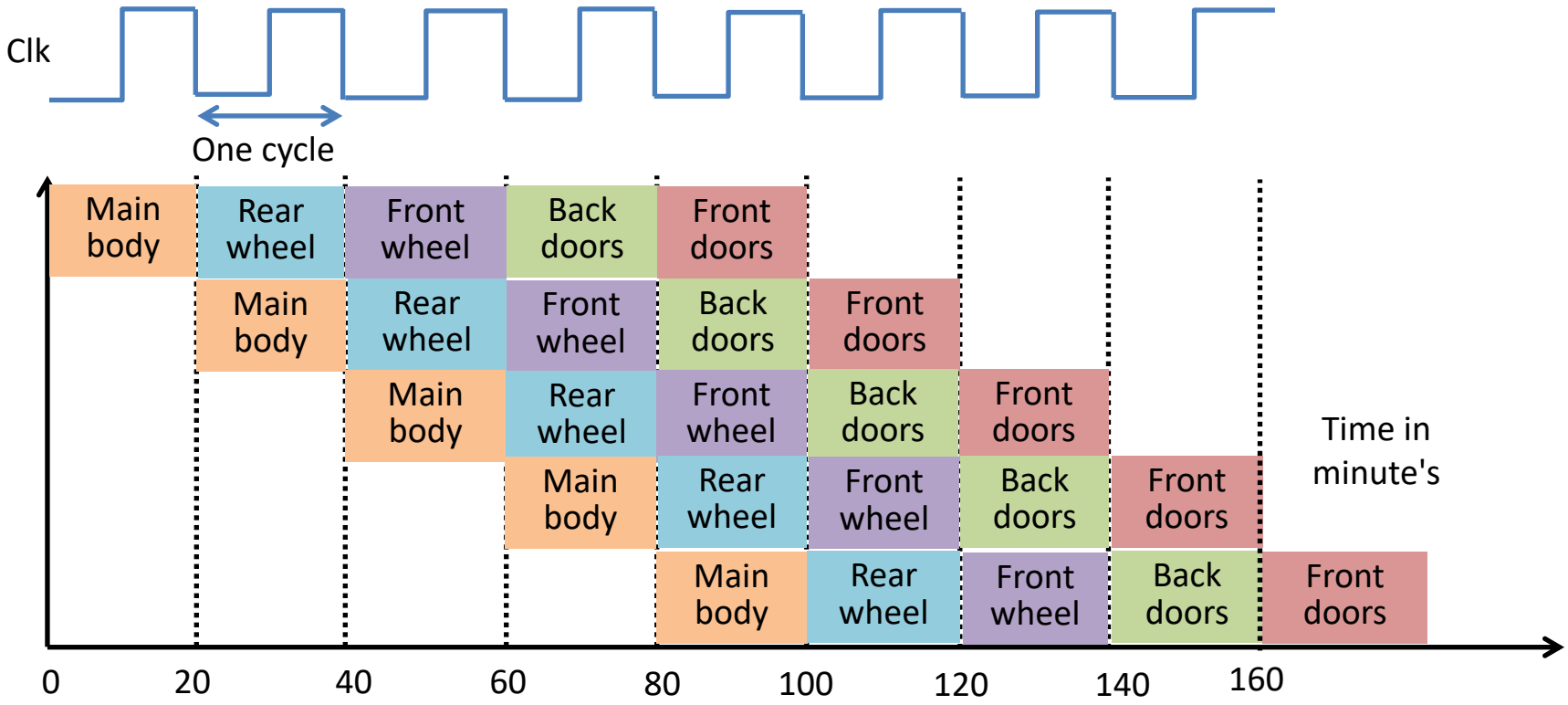
Advantage: Low CPI(1)

Disadvantage: Long clock period(to accommodate slowest instruction)

Pipelining Analogy

- Pipelined: Building a car

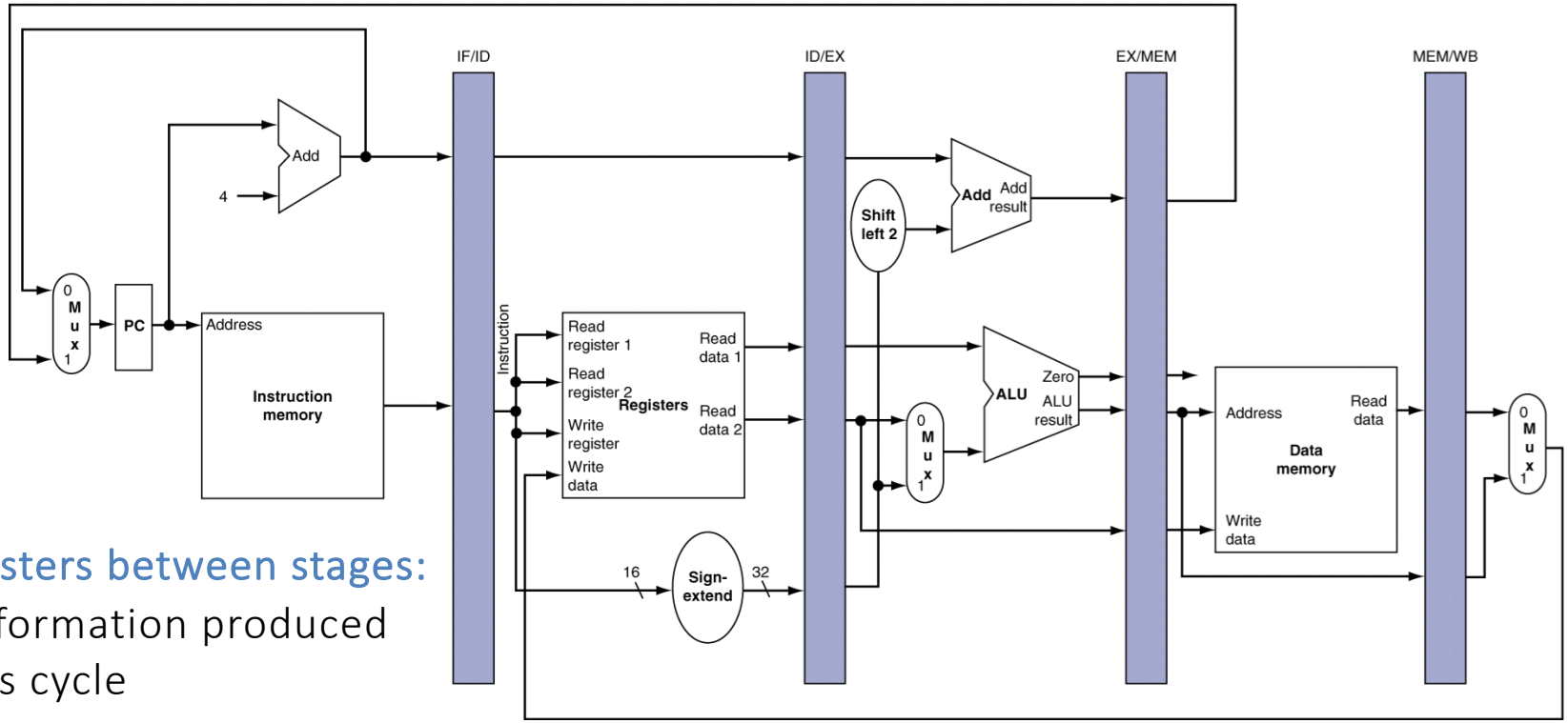
- **Pipelining** allows multiple sub-tasks to be carried out simultaneously using independent resources
- Need to **balance** time taken by each sub-task



Pipelining Analogy

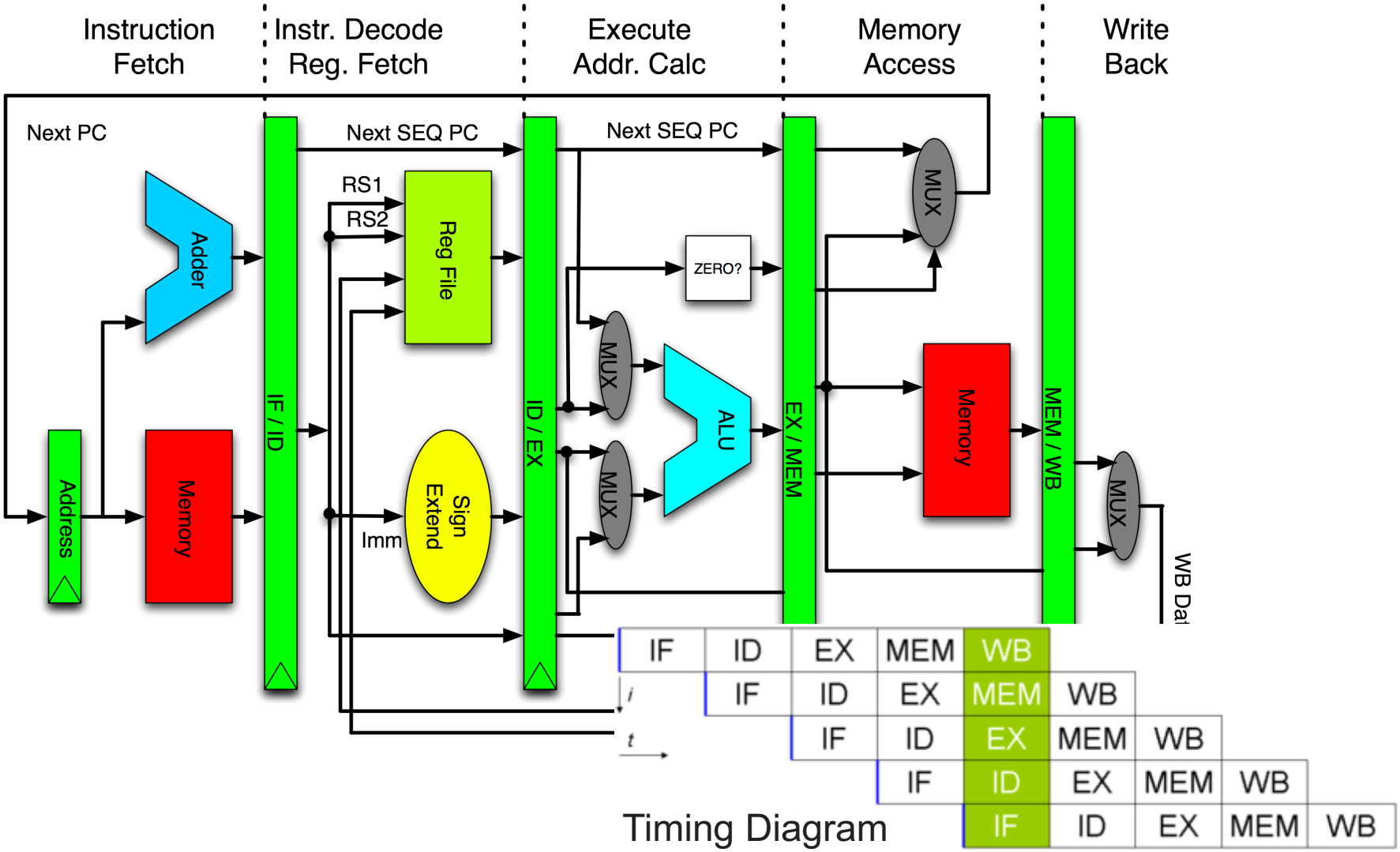
- MIPS Pipelined
- Five stages, one step per stage:

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

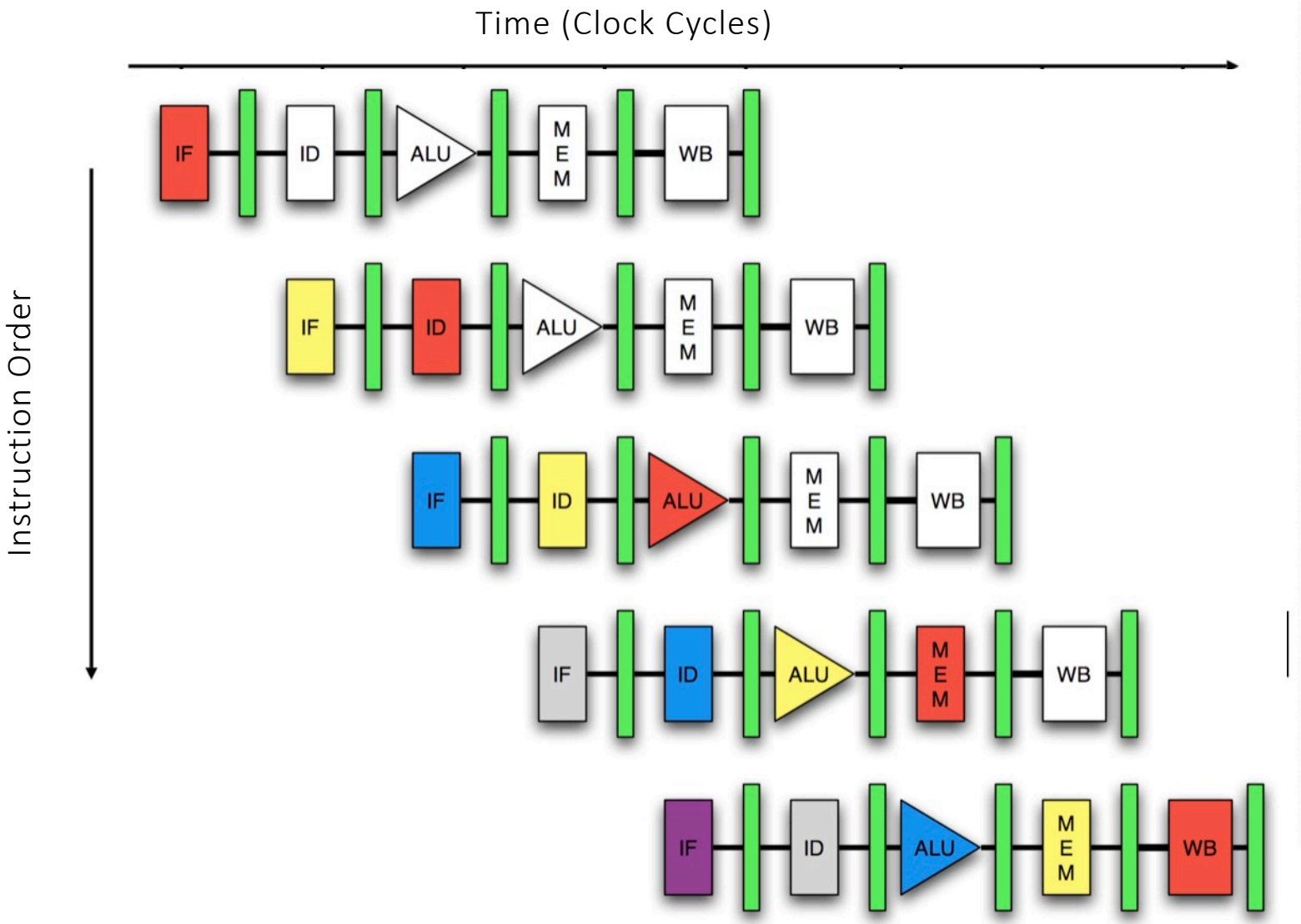


Need Registers between stages:
To hold information produced in previous cycle

PIPELINED PROCESSOR

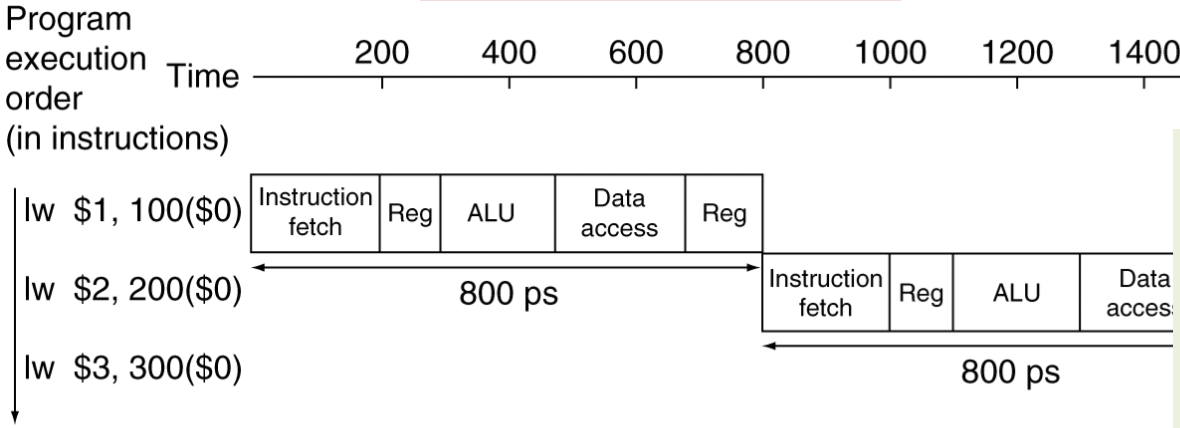


PIPELINED PROCESSOR

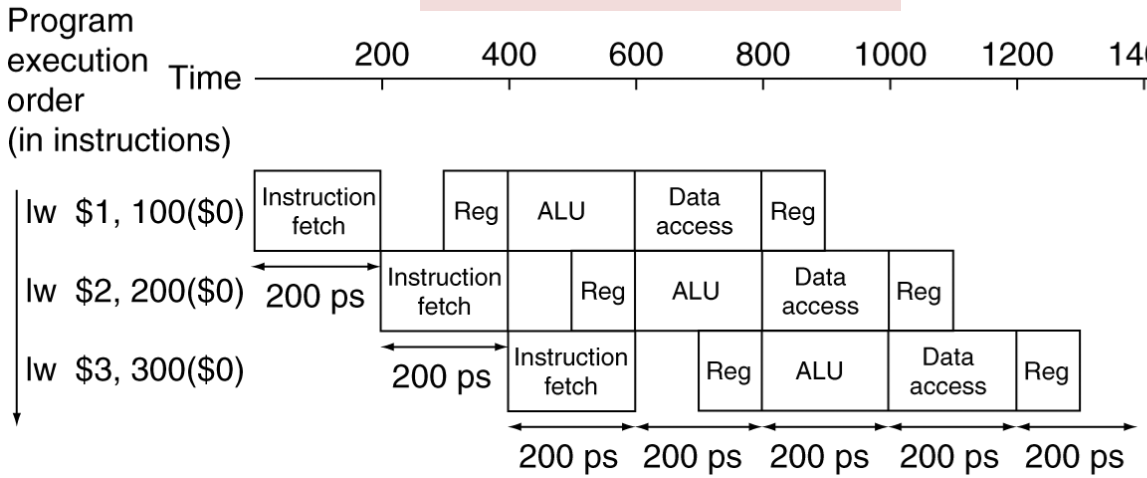


Pipelining Performance

Single-cycle (Tc = 800ps)



Pipelined (Tc = 200ps)



If all stages are balanced:

-- All take the same time

Time between instructions$_{Pipelined}$

=Time between instructions$_{NonPipelined}$/# stages

If not balanced, speedup is less

Speedup is due to the increased Throughput:

Latency (time for each instruction) does not increase

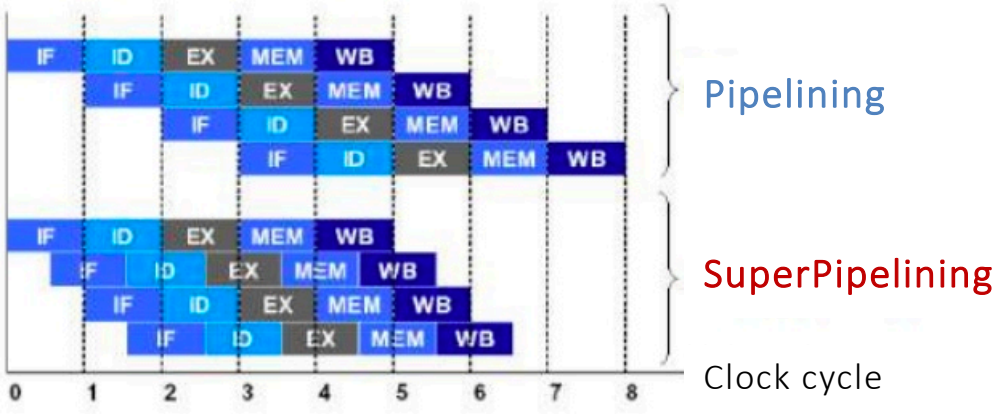
SUPERPIPELINED PROCESSORS

- Increase the depth of the pipeline leading to shorter clock cycles

- The higher the degree of superpipelining, the more forwarding/hazard hardware needed, more pipeline latch overhead
  - (i.e., pipeline latch accounts for a larger and larger percentage of the clock cycle time)

Super pipeline Performance

- The performance is shown below in the figure:



Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - -- All instructions are 32-bits
    - ✓ Easier to fetch and decode in one cycle
  - -- Few and regular instruction formats
    - ✓ Can decode and read registers in one step
  - -- Load/store addressing
    - ✓ Can calculate address in 3th stage, access memory in 4th stage
  - -- Alignment of memory operands
    - ✓ Memory access takes only one cycle

Quiz

- Single-Cycle versus Pipelined Performance

In this quiz (example), we limit our attention to eight instructions: load word (lw), store word (sw), add (add), subtract (sub), AND (and), OR (or), set less than (slt), and branch on equal (beq).

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Compare the average time between instructions of a single-cycle implementation (all instructions take one clock cycle) to a pipelined implementation.

Answer

- Please reply your answers to TA before this Friday (Mar 5, 2021)
- More detailed description will be better in your answers

Instruction Level Parallelism (ILP)

- Goals:
  - What is instruction-level parallelism (ILP)?
  - What do processors do to extract ILP?
  - How do processors conduct ILP?

- Why do processors get faster?
  1. More parallelism (or more work per pipeline stage): Get Wider
     -- Fewer clocks/instruction [more instructions/cycle]
  2. Deeper pipelines: Get Deeper
     -- Fewer gates/clock
  3. Transistors get faster (Moore's Law): Get Faster
     -- Fewer ps/gate

Instruction Level Parallelism

- Parallel execution of instructions requires 3 major tasks

1. Which instructions should be executed in parallel?

   -- Checking the data dependence between instructions to identify the instructions which can be grouped together for parallel execution.

2. Where to be executed?

   -- Assigning instructions to different functional units in processor.

3. When to be executed?

   -- Determining when instruction execution is to be initiated.

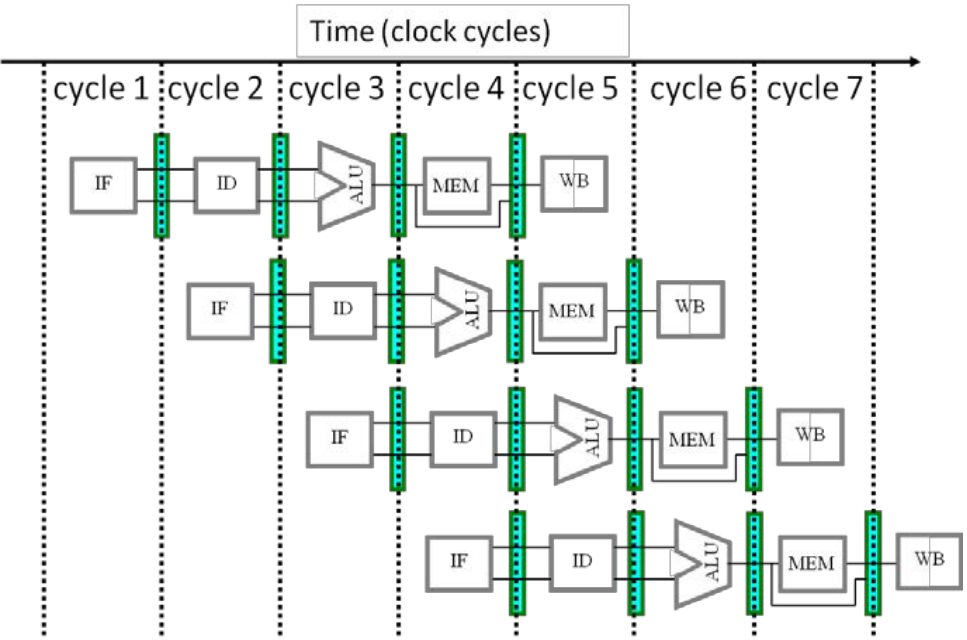
Instruction Level Parallelism (ILP)

- Extracting yet more performance
  1. Increase the depth of the pipeline to increase the clock rate – Superpipelining
     - How does this help performance?
  2. Fetch (and execute) more than one instruction at one time (expand every pipeline stage to accommodate multiple instructions) – Multiple-Issue
     - How does this help performance? (impact in performance equation)
     - Today's topic! $\frac{Second}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Second}{Cycle}$
  3. Launching multiple instructions per stage allows instruction execution rate, CPI, to be less than 1
     - Instead we use IPC (instructions per clock cycle): e.g., a 3 GHz, 4-way multiple-issue processor can execute at a peak rate of 12 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4

Sequential Execution	Instruction-Level Parallelism
1. a = 10 + 5 2. b = 12 + 7 3. c = a + b Instructions: 3 Cycles: 3	1.A. a = 10 + 5 1.B. b = 12 + 7 2. c = a + b Instructions: 3 Cycles: 2 (-33%)

ILP and Multi-Issue Processors

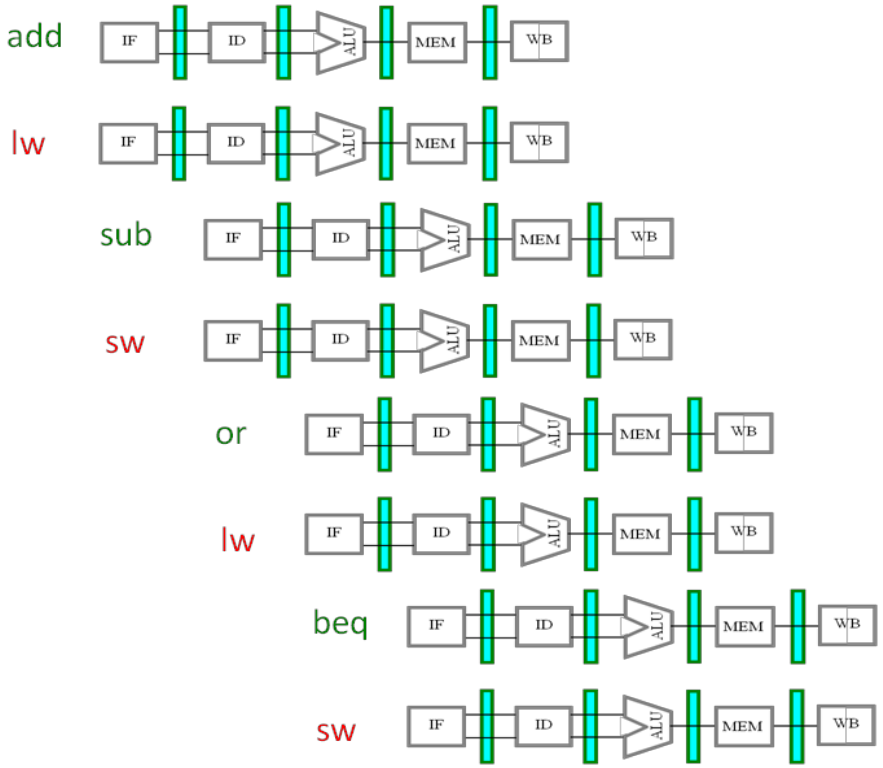
- 1 instruction issued in a clock cycle
  - -- Single-Issue Processor



Pipelined processor

Scalar processor

- 2 instruction issued in a clock cycle
  - -- Multi-Issue Processor



Super scalar processor

2-way superscalar

➔2 or more instructions are issued in a clock cycle

Instruction Level Parallelism

- Two approaches to achieve ILP

1. Hardware Approach: Superscalar Processor

   --P5 Pentium, the first superscalar X86 processor

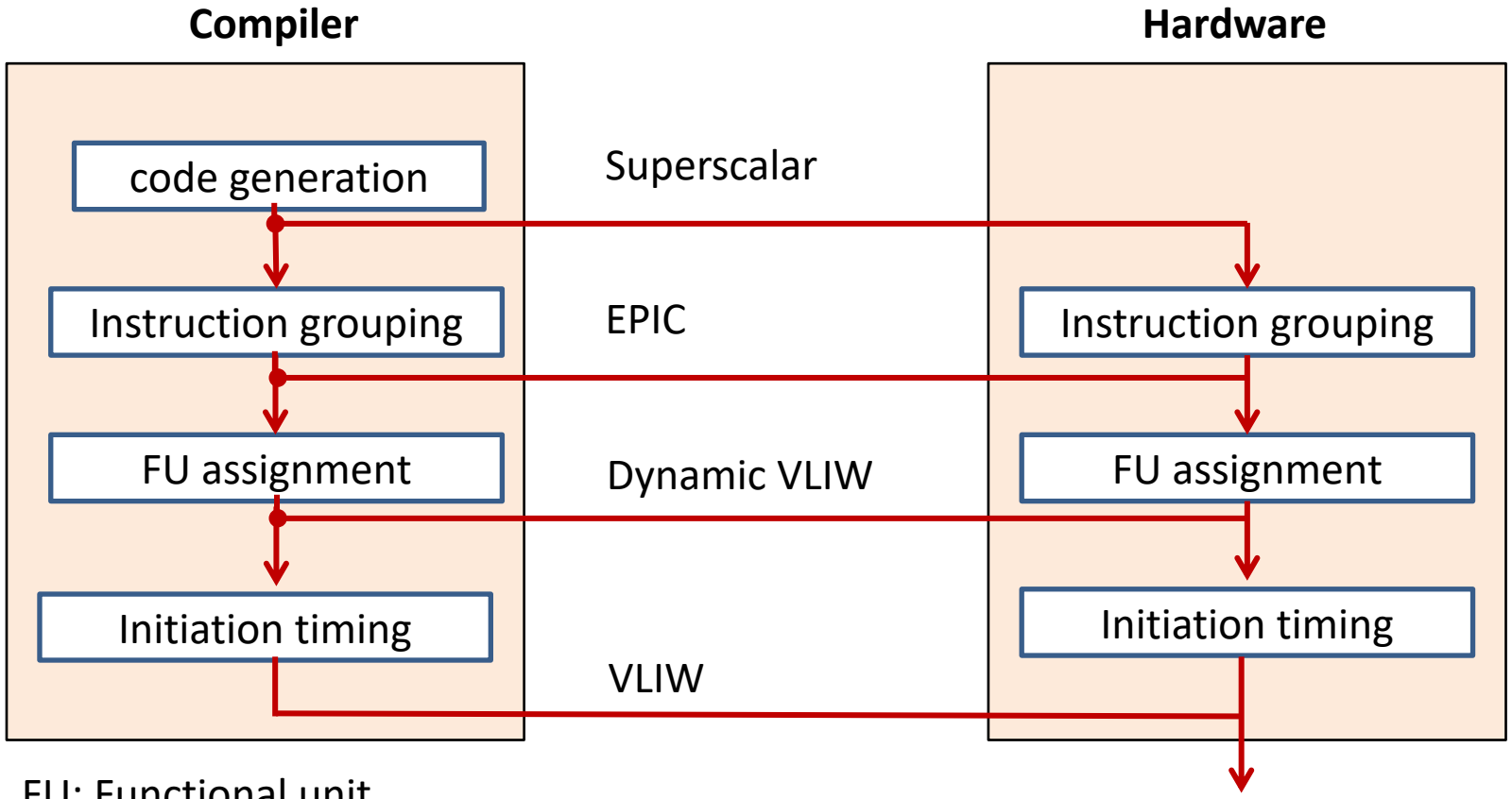
   -- Most general-purpose CPU's developed since 1998 are superscalar

2. Software (or Compiler-based) Approach: Very Long Instruction Word (VLIW) processor

   -- VLIW CPU's contain multiple RISC like Functional Units(FUs). Typically have 4 to 8 FUs.

Instruction Level Parallelism

- Hardware and Software Approaches

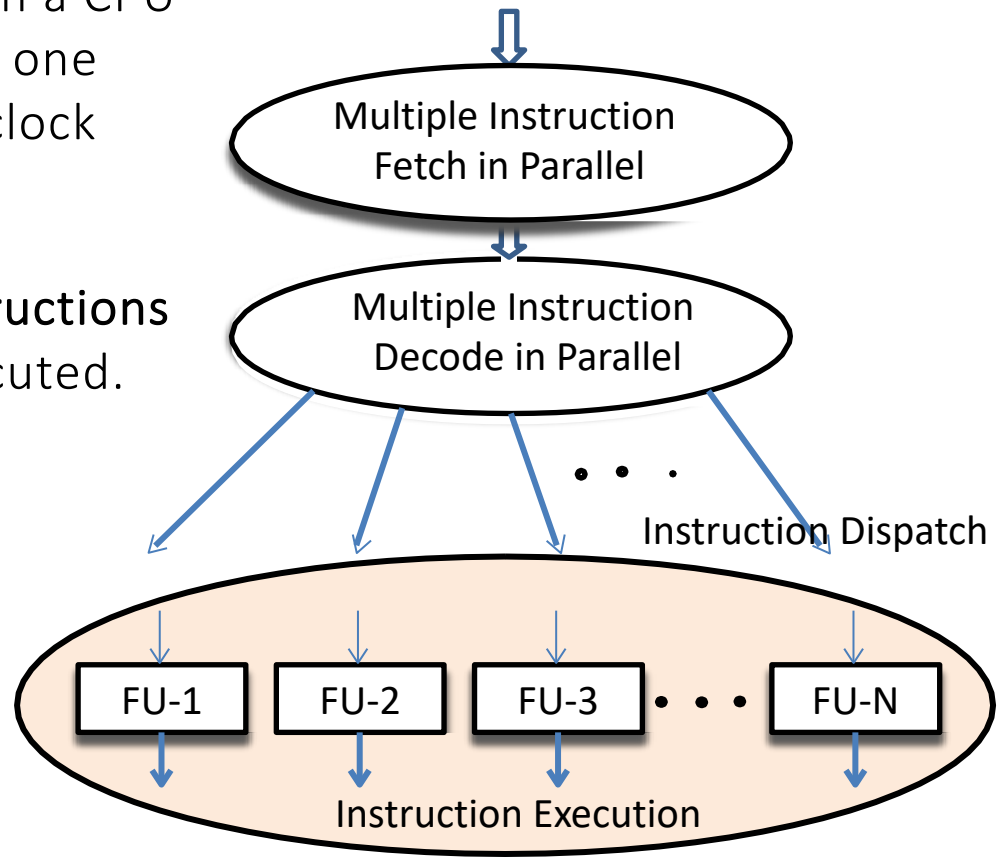


FU: Functional unit
EPIC: Explicitly parallel instruction computing

Instruction Level Parallelism

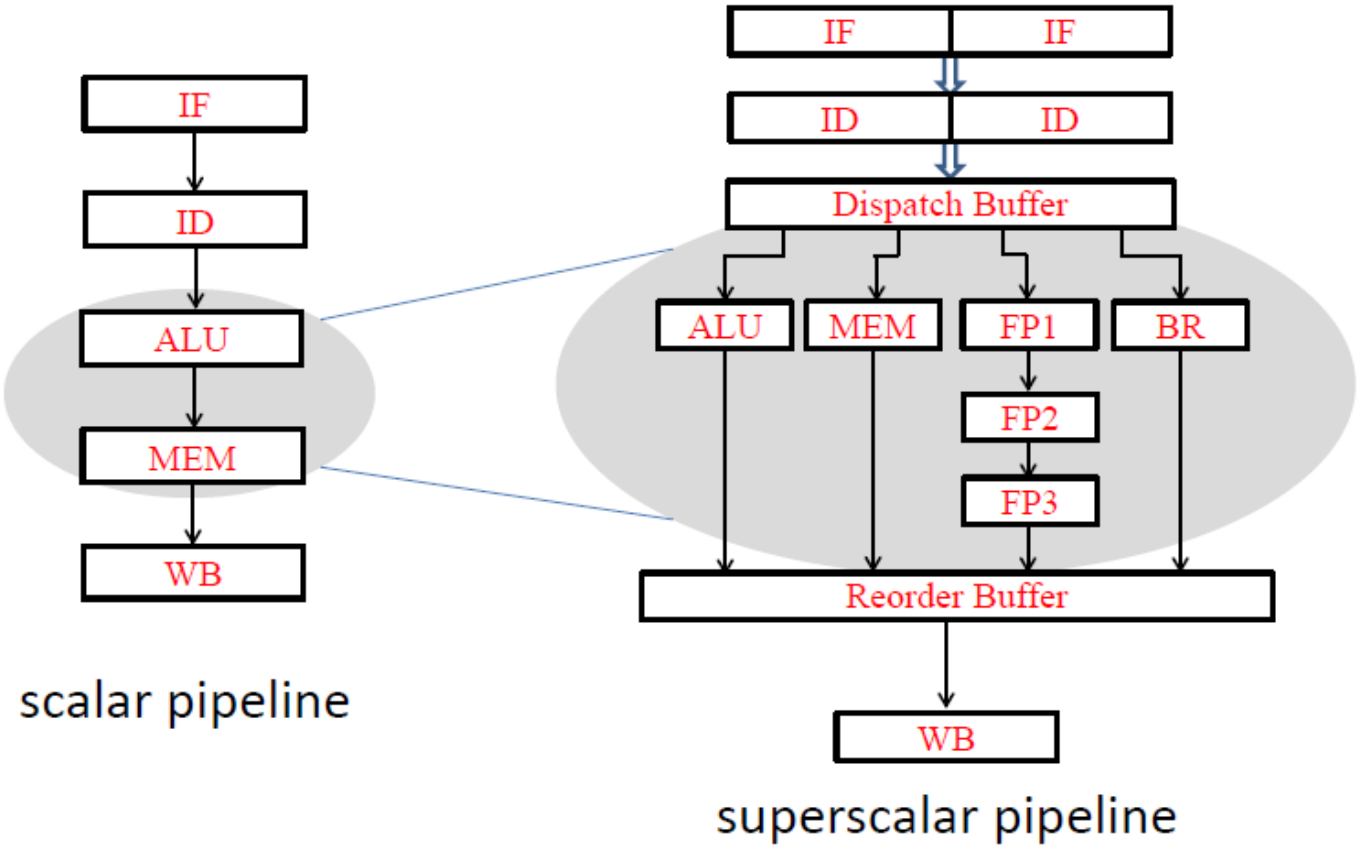
- Concept of Superscalar Processing and the Requirements

  - **Multiple functional units (FU)** in a CPU are used to execute more than one instructions concurrently in a clock cycle.
  - **More than 1 independent instructions** need to be available to be executed. (Makes use of ILP)
  - **Use specialized hardware** to issue multiple independent instructions that can be executed simultaneously.



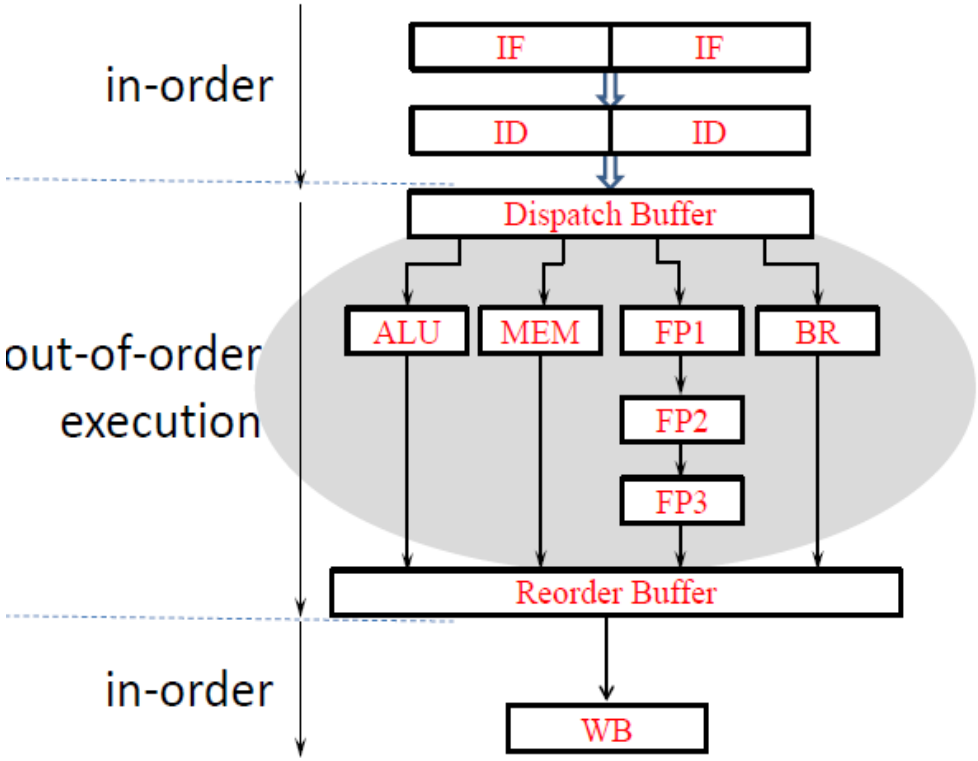
Instruction Level Parallelism

- Scalar and Superscalar Processor



Instruction Level Parallelism

- Out-of-order execution in Superscalar pipeline
  - Instructions are fetched in compiler-generated order (in-order).
  - Instruction completion may be also in order.
  - Instructions are executed in some other order: independent instructions behind a stalled instruction can be executed prior to the stalled instruction.
  - Dynamically scheduled: the order of execution of instructions is changed.



Instruction Level Parallelism

- Methods to extract more parallelism
  - Instruction reordering and out-of-order execution
  - Speculative execution with dynamic scheduling
  - Loop unrolling

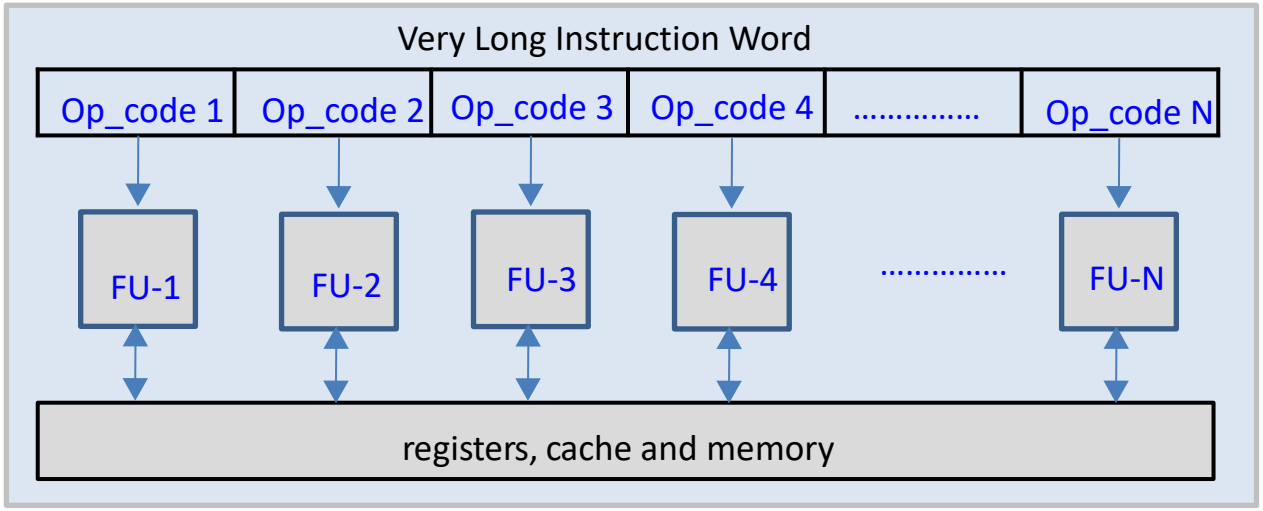
-- **Instruction Reordering**: Change the order of execution of instruction if it does not violate the data dependence.

-- **Speculative Execution**: To execute an instruction without exactly knowing if that need to be executed: ahead of branch outcome.

-- **Dynamic Scheduling**: Execute instructions as soon as dependencies are satisfied and functional units are available.

Instruction Level Parallelism

- VLIW processing strategy
  - Pack multiple independent operations into one instruction.
  - VLIW processor needs a compiler to break the program instructions down into basic operations that can be performed by the processor in parallel.
  - These operations are put into a very long instruction word which the processor (consisting of multiple functional units (FU)) gets executed in appropriate FUs.
  - There must be enough parallelism in the program code to fill the available slots, to reduce the CPI significantly.



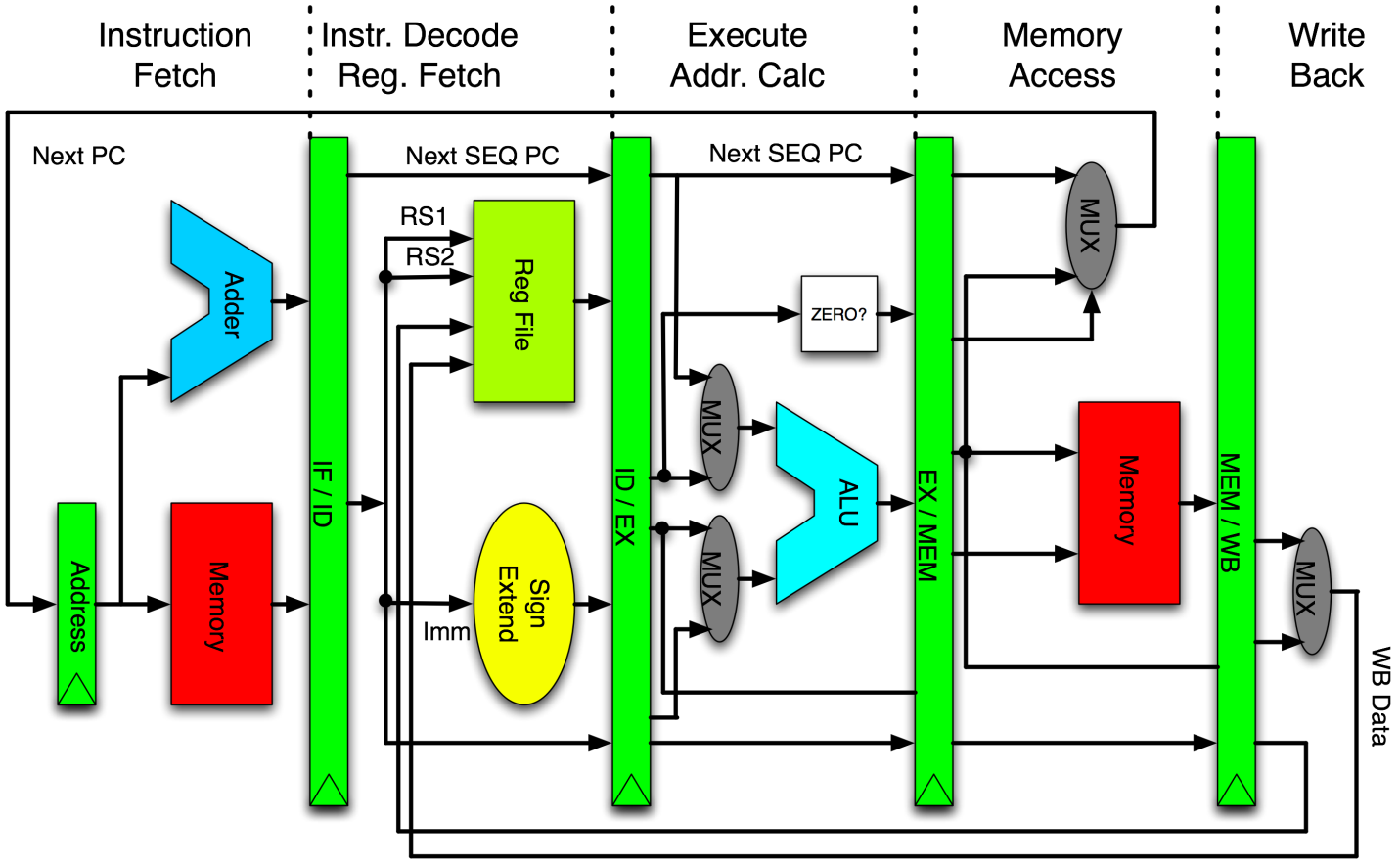
- Multiple operation execution.
- Many functional units: each is executing its own instruction.
- Typically maximum of 4 or 8 instructions per cycle are issued.

Instruction Level Parallelism

- Advantages and Disadvantages of VLIW processing
  - **Compiler prepares fixed packets of multiple operations:** give complete plan of execution.
    - Dependencies are determined by compiler and that is used to schedule according to the latencies of corresponding functional units.
    - Functional units are assigned by compiler, they correspond to the position within the instruction packet or instruction slot.
    - Compiler produces fully-scheduled, hazard-free code => hardware need not check dependencies for scheduling.
  - **Compatibility across implementations is a major problem.**
    - VLIW code cannot run properly with different number of functional units or functional units with different latencies.
    - Unscheduled events (e.g., cache miss) can stall the entire processor.
  - **Code density is an important concern.**

Pipeline Hazards

PIPELINING – SUPERSCALAR - PIPELINE HAZARDS



[Pipelining & Superscalar & Pipeline Hazard]

Review: Pipeline Hazards

Hazards are bad: reduce the amount of achievable machine parallelism and keep us from achieving all the Instruction-Level Parallelism (ILP) in instruction stream

1. Structural hazards

- What? – Resource conflict when HW cannot support all possible combination of instructions simultaneously/overlapped instructions
- How to eliminate? – ***Duplicating resources***

2. Data hazards

- What? – An instruction depends on result of a previous instruction in a way that is exposed by overlapping of instruction in pipeline; need to wait for previous instruction to complete read/write
- How do we eliminate? – ***Forwarding/Bypassing***

3. Control hazards – beq, bne, j, jr, jal

- What? – Pipeline of branches and other instructions that change PC (Program Counter); deciding on control action depends on previous instruction
- How do we eliminate? – ***Insert pipeline bubble***

Review: Pipeline Hazards

- How to deal with hazards?
  - Often, pipeline must be stalled. **Pipeline Stall? Bubble?**
  - Stalling pipeline usually lets some instructions in pipeline proceed, another/others wait for data, resource, etc.

structural hazards

Instr	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Stall				bubble	bubble	bubble	bubble	bubble		
Instr i+3										
Instr i+4										

Data hazards

Instr	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	bubble	EX	MEM	WB			
Instr i+2			IF	bubble	ID	EX	MEM	WB		
Instr i+3				bubble	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

Review: Pipeline Hazards

- Performance of Pipelines with stalls
  - Stalls impede the progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
  - Pipelining can be viewed to decrease CPI or clock cycle time for instruction

$$CPI_{pipelined} = Ideal\ CPI + Pipeline\ stall\ cycles\ per\ instruction$$

$$= 1 + Pipeline\ stall\ cycles\ per\ instruction$$

$$Speedup = \frac{CPI_{unpipelined}}{1 + Pipelining\ stall\ cycles\ per\ instruction}$$

Review: Pipeline Hazards

- **Performance of Pipelines with stalls**
  - If the unpipelined CPI is equal to the depth of the pipeline:

$$Speedup = \frac{Pipeline\ Depth}{1 + Pipelining\ stall\ cycles\ per\ instruction}$$

$$Speedup from\ Pipeline = \frac{CPI\ unpipelined}{CPI\ pipelined} \times \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

$$= \frac{1}{1 + pipeline\ stall\ cycles\ per\ instruction} \times \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

  - Since:

$$Clock\ cycle\ pipelined = \frac{Clock\ cycle\ unpipelined}{Pipeline\ depth} ==> Pipeline\ depth = \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

  - Therefore, this will lead to the following:

$$\boldsymbol{Speedup from\ Pipeline == \frac{1}{1 + pipeline\ stall\ cycles\ per\ instruction} \times Pipeline\ depth}$$

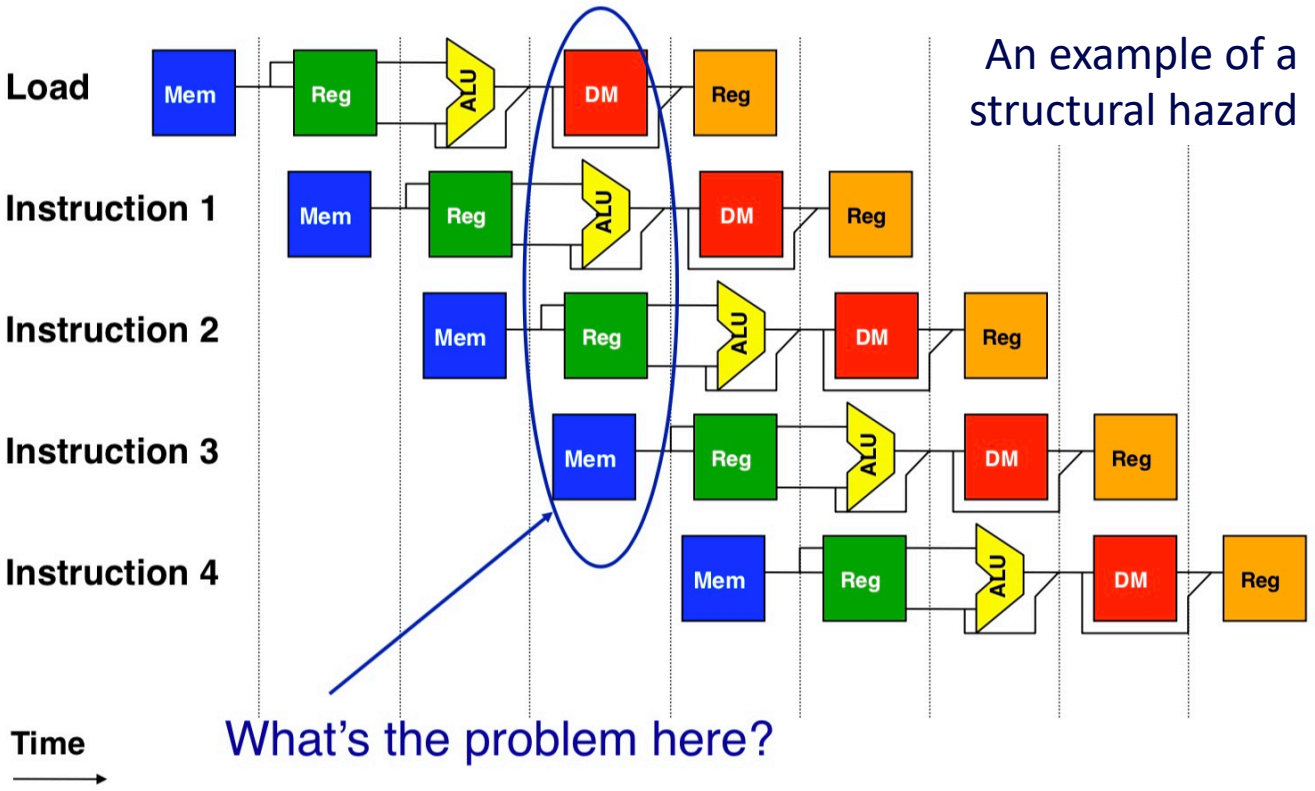
I. Structural hazards

PIPELINE HAZARDS - STRUCTURAL HAZARDS

- If not all possible combinations of instructions can be executed, structural hazards occur
- Avoid Structural hazards by duplicating resources

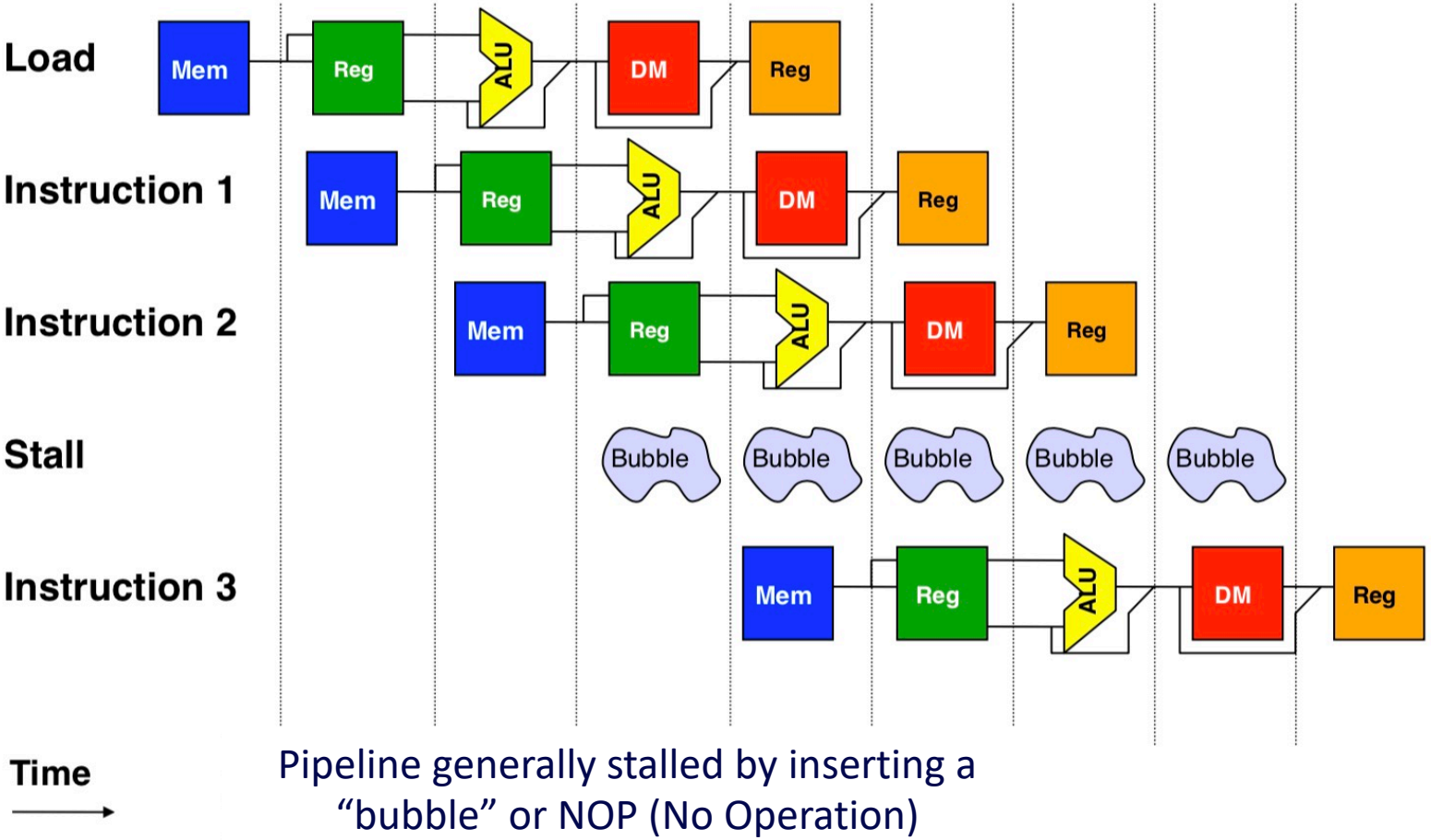
  -- E.g., an ALU to perform an arithmetic operation and an adder to increment PC

- Pipelines stall result of the hazards, the CPI increased from usual "1"



PIPELINE HAZARDS - STRUCTURAL HAZARDS

- How is it resolved?



Pipeline generally stalled by inserting a "bubble" or NOP (No Operation)

PIPELINE HAZARDS - STRUCTURAL HAZARDS

- Or alternatively…

Clock Number →

Inst. #	1	2	3	4	5	6	7	8	9	10
LOAD	IF	ID	EX	MEM	WB					
Inst. i+1		IF	ID	EX	MEM	WB				
Inst. i+2			IF	ID	EX	MEM	WB			
Inst. i+3				stall	IF	ID	EX	MEM	WB	
Inst. i+4						IF	ID	EX	MEM	WB
Inst. i+5							IF	ID	EX	MEM
Inst. i+6								IF	ID	EX

- LOAD instruction "steals" an instruction fetch cycle which will cause pipeline to stall
- Thus, no instruction completes on clock cycle 8

- What is the realistic solution?

  -- Answer: Add more hardware

Especially for the memory access example, i.e., common case, CPI degrades quickly from our ideal "1" for even the simplest of cases

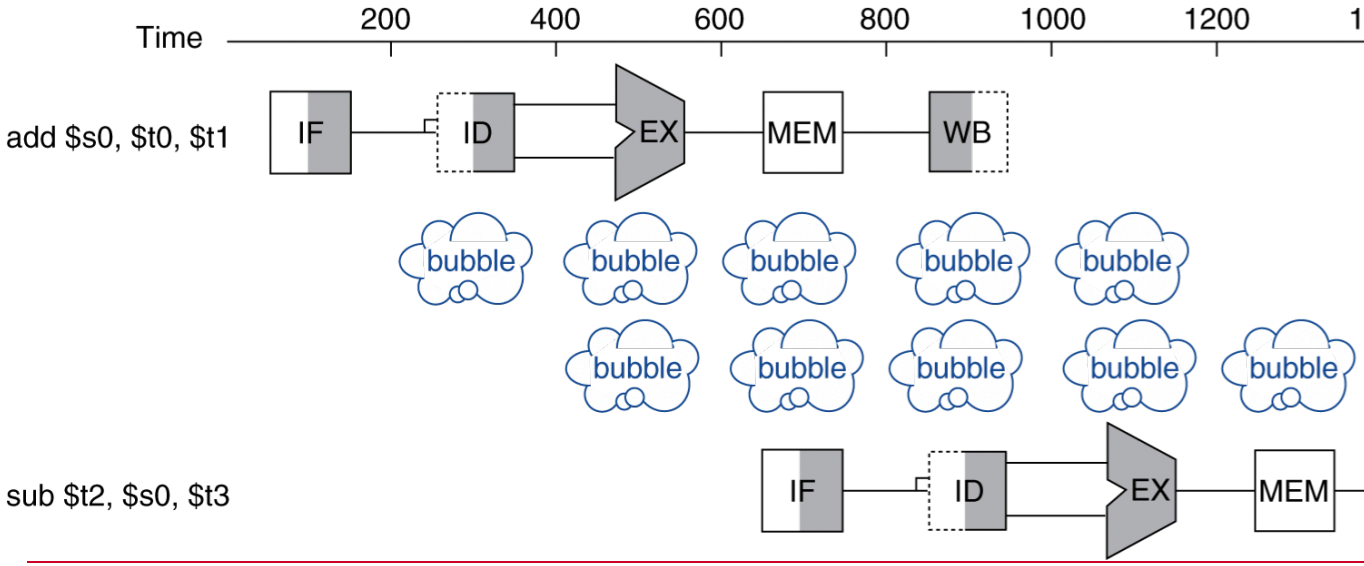
II. Data hazards

PIPELINE HAZARDS - DATA HAZARDS

Data hazards exit because of pipelining

- Why do they exist?
  - An instruction depends on completion of data access by a previous instruction; Pipelining changes when data operands are read, written
  - Order differs from order seen by sequentially executing instructions on un-pipelined machine

```
-- ADD $s0, $t0, $t1
-- sub $t2, $s0, $t3
```



- All instructions after ADD use the result of ADD
- ADD writes the register in WB but SUB needs it in ID

This is the data hazards

Pipeline Hazards - Data hazards

- Data hazards specifics

- There are actually 3 different kinds of data hazards
  - Read After Write (RAW)
    $Instr_J$ tries to read operand before $Instr_I$ writes it

    ```
    I: add r1,r2,r3
    J: sub r4,r1,r3
    ```

    → Results from an actual need for communication

    – Caused by a "**dependence**" (in compiler nomenclature).
  - Write After Read (WAR)
    $Instr_J$ writes operand *before* $Instr_I$ reads it

    ```
    I: sub r4,r1,r3
    J: add r1,r2,r3
    ```

    – Called an "**anti-dependence**" by compiler writers.
  - Write After Write (WAW)
    $Instr_J$ writes operand *before* $Instr_I$ writes it.

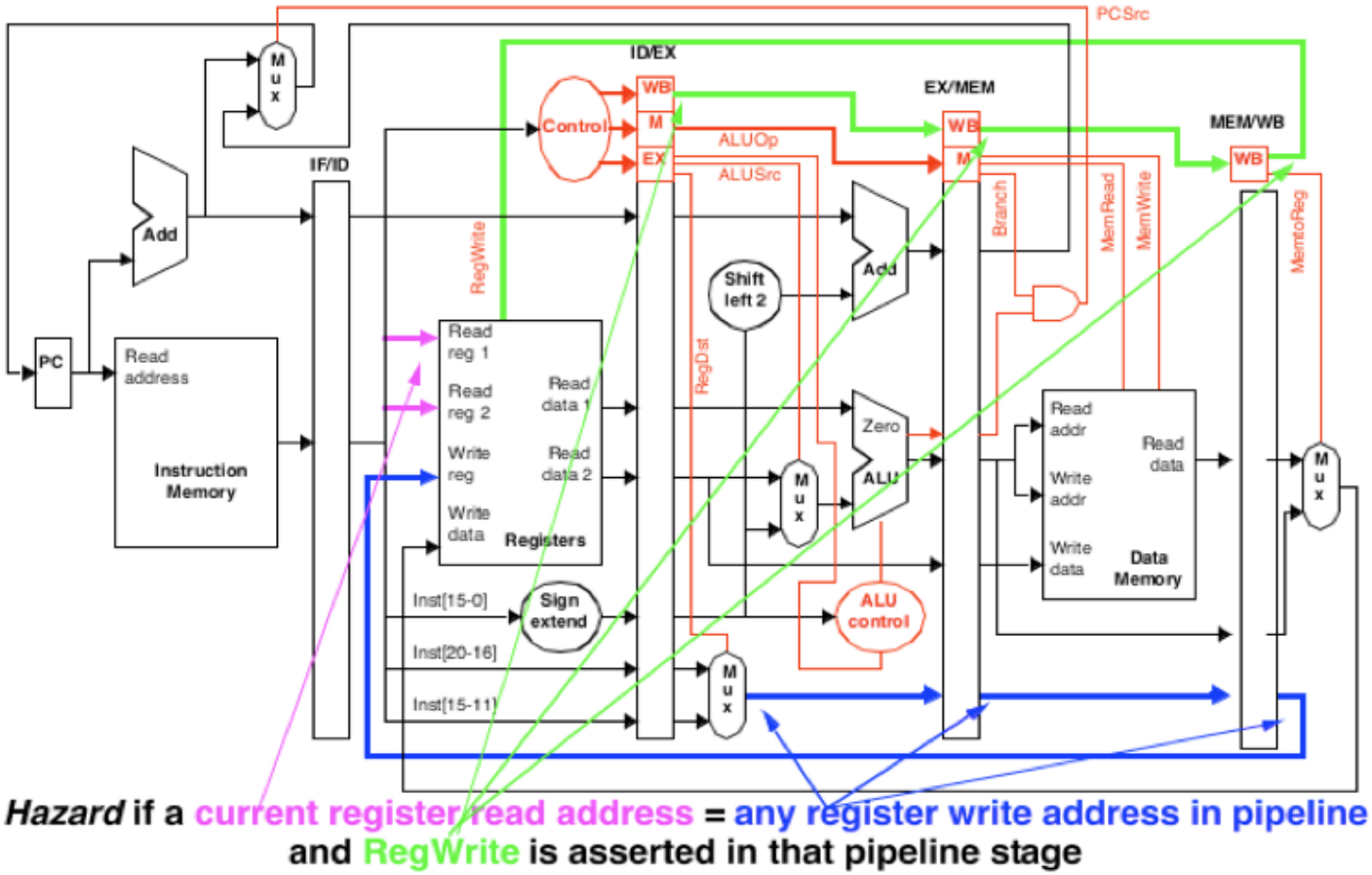
    ```
    I: sub r1,r4,r3
    J: add r1,r2,r3
    ```

    – Called an "**output dependence**" by compiler writers

- Result from the reuse of name "r1",
- Can't happen in MIPS 5 stage pipeline

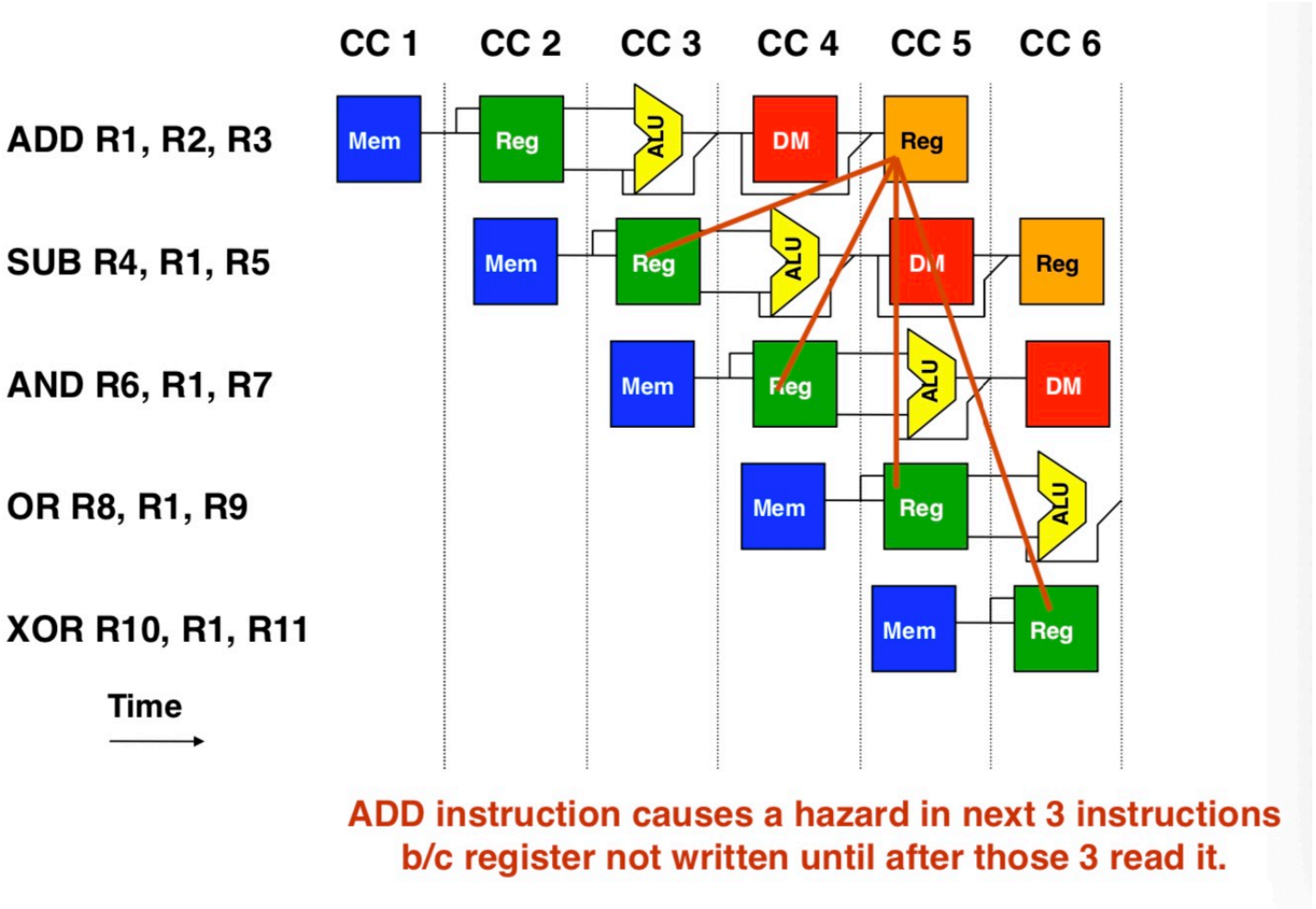
PIPELINE HAZARDS - DATA HAZARDS

- Detecting Data hazards specifics



PIPELINE HAZARDS - DATA HAZARDS

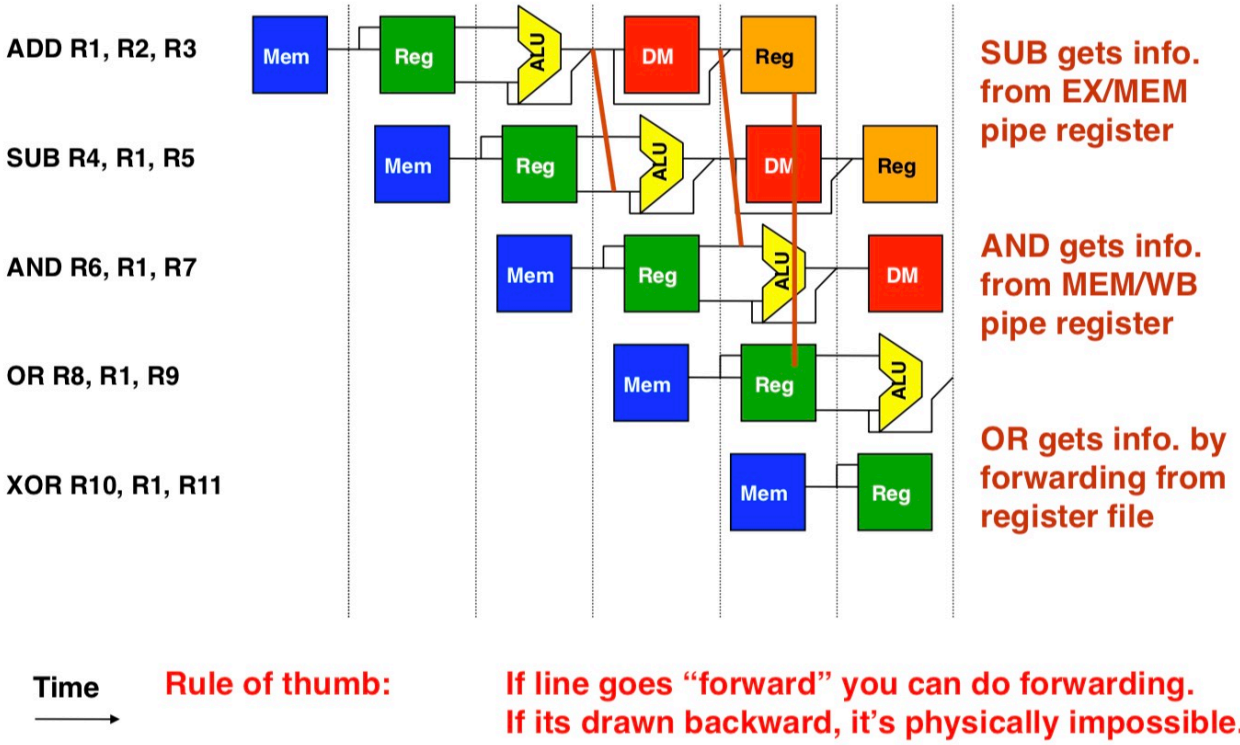
An example of a Data hazard



PIPELINE HAZARDS - DATA HAZARDS

- Handling Data Hazard: Forwarding
  - Problem illustrated on the example of data hazard can actually be solved by relatively easily – with forwarding

When can we forwarding?

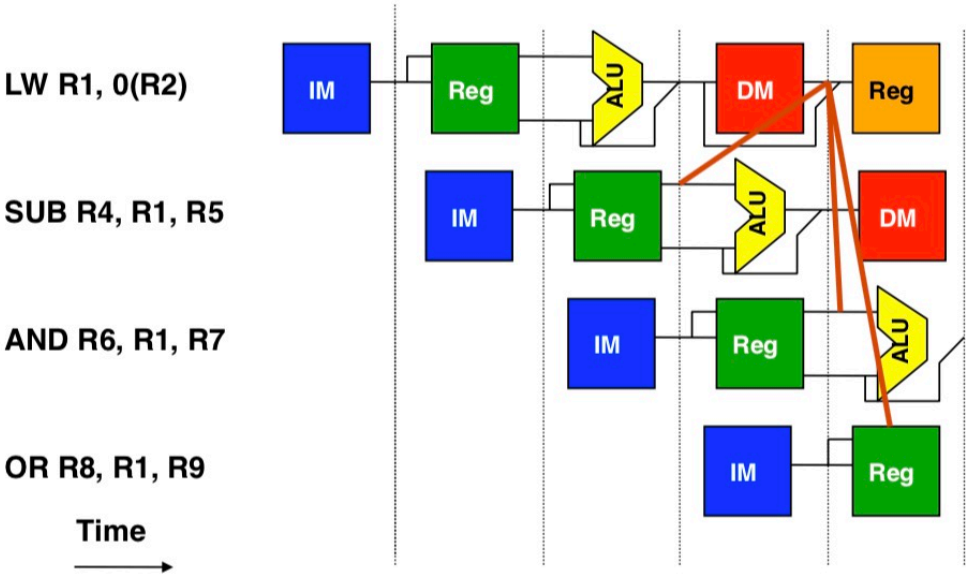


- Generally:

1. Forwarding occurs when a result is passed directly to functional unit that requires it
2. Result goes from output of one unit to input of another

PIPELINE HAZARDS - DATA HAZARDS

- Forwarding doesn't always work



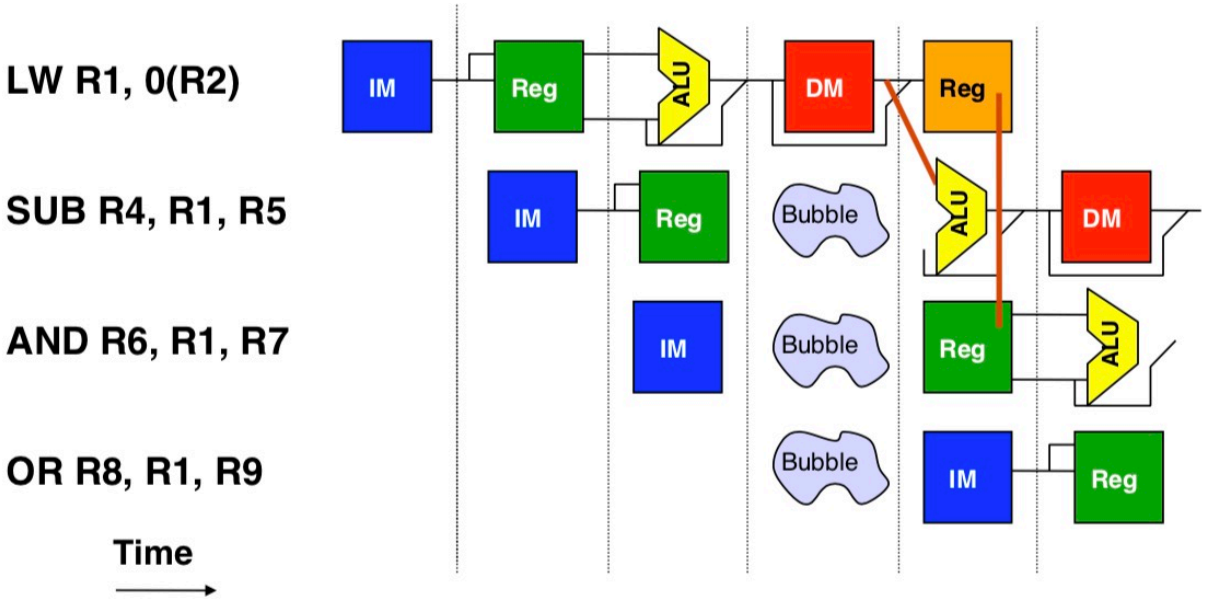
Load has a latency that forwarding can't solve.

Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).

Can't get data to subtract b/c result needed at beginning of CC #4, but not produced until end of CC #4.

PIPELINE HAZARDS - DATA HAZARDS

- Handling Data Hazard: Insert Bubble or nops in code at compile time



Insertion of bubble causes # of cycles to complete this sequence to grow by 1

PIPELINE HAZARDS - DATA HAZARDS

Data hazards and the compiler

- Compiler should be able to help eliminate some stalls caused by data hazards

-- i.e., compiler could not generate a LOAD instruction that is immediately followed by instruction that uses result of LOAD's destination register

What about the control logic?

- For MIPS integer pipeline, all data hazards can be checked during ID phase of pipeline
- If data hazard, instruction stalled before its issued
- Whether forwarding is needed can be determined at this stage, control signals set
- If hazard detected, control unit of pipeline must stall pipeline and prevent instructions in IF, ID from advancing

Situation	Example	Action

No Dependence	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	LW R1, 45(R2)
	ADD R5, R1, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX

Dependence overcome by forwarding	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R1, R7
	OR R9, R6, R7
		Comparators detect the use of R1 in SUB and forward the result of LOAD to the ALU in time for SUB to begin with EX

Dependence with accesses in order	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R1, R7
		No action is required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Overcome Data hazards: Dynamic Scheduling

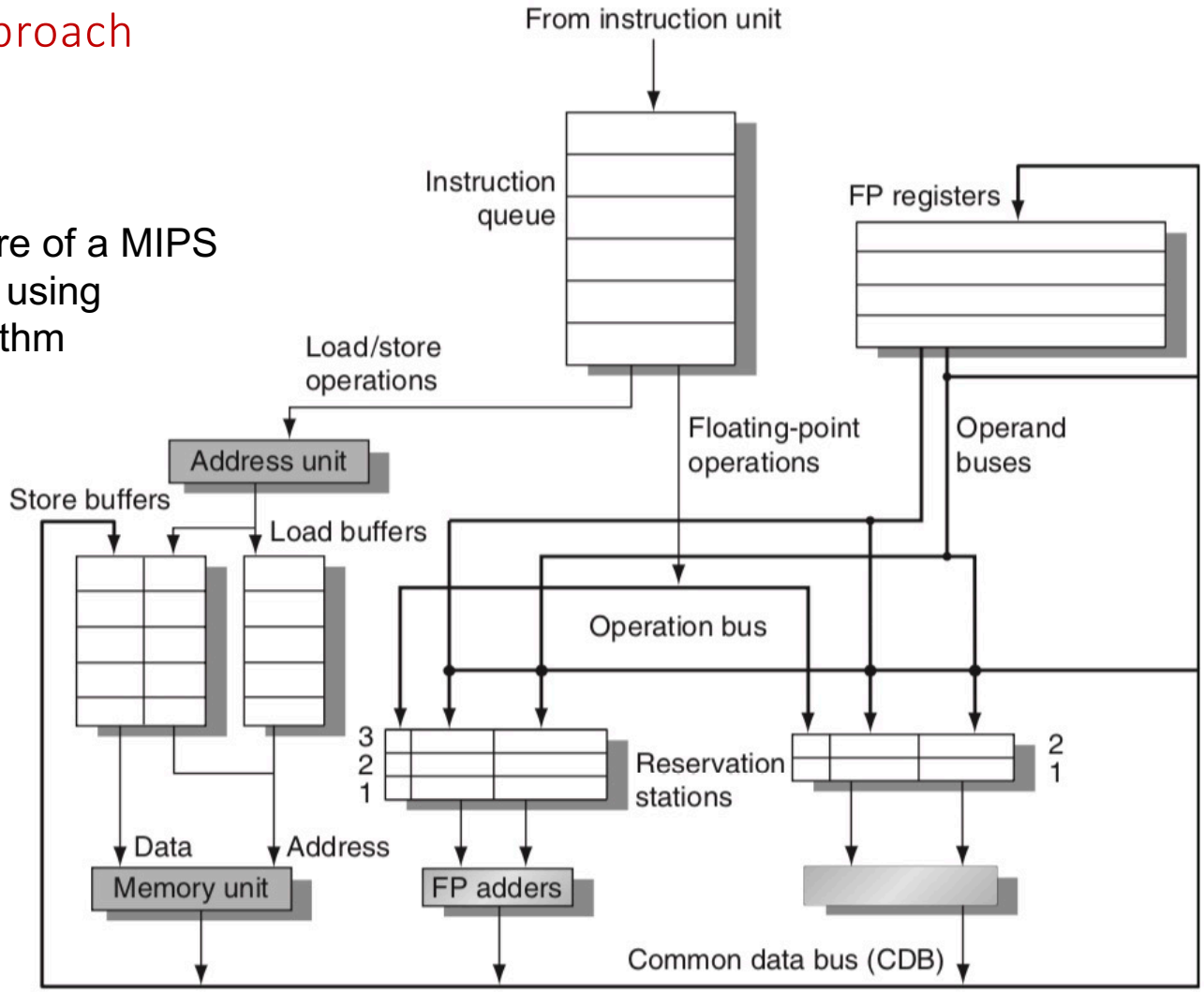
- Dynamic scheduling
  - It allows code that was compiled with one pipeline to run on another different pipeline
  - It can handle some cases when dependences are unknown at the compile time
  - It allows the processor to tolerate unpredictable delays

- Dynamic Scheduling Using Tomasulo's Approach

Overcome Data hazards: Dynamic Scheduling

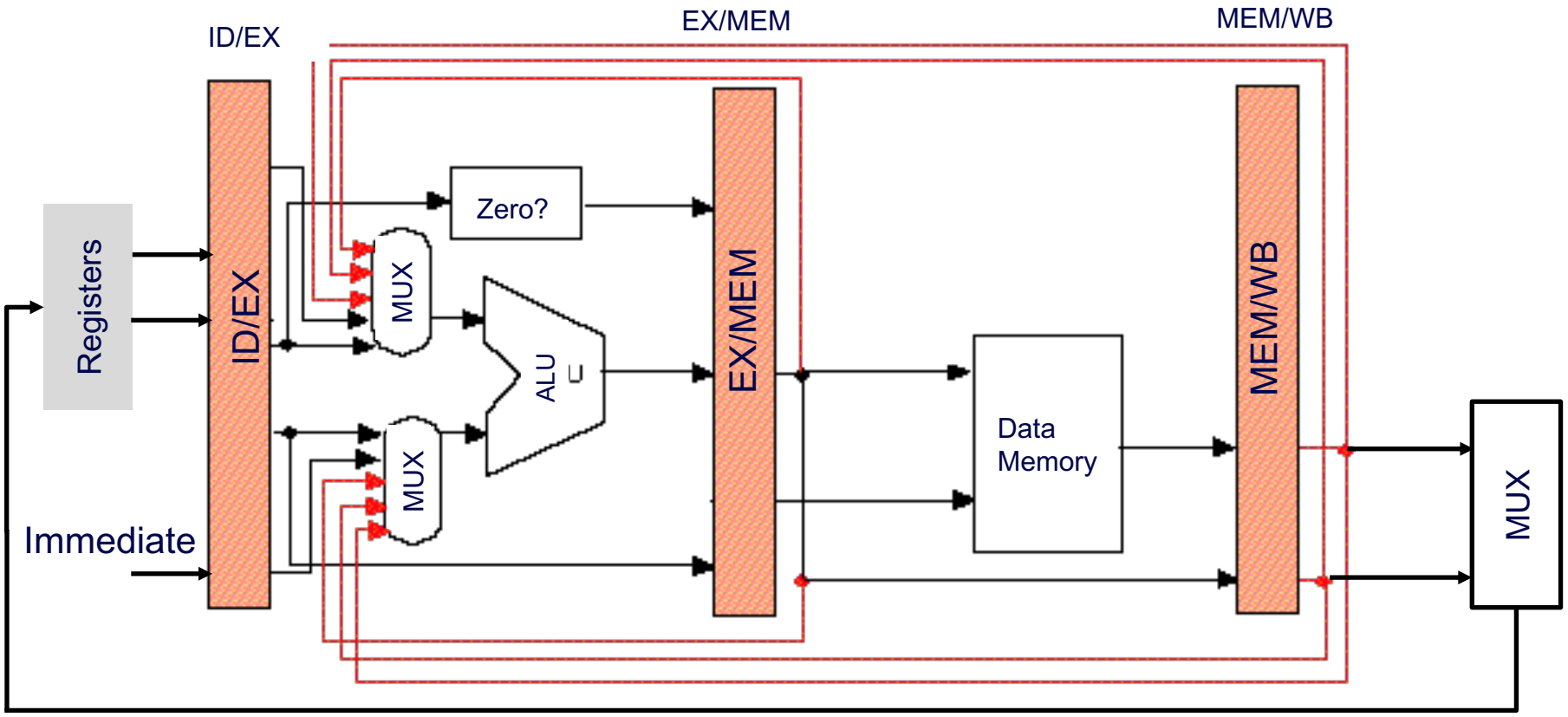
- Tomasulo's Approach

The basic structure of a MIPS floating-point unit using Tomasulo's algorithm



Pipeline Hazards - Data hazards

- Hardware change for Forwarding



Send output of memory back to ALU too…

Review: Pipeline Hazards

Hazards are bad: reduce the amount of achievable machine parallelism and keep us from achieving all the Instruction-Level Parallelism (ILP) in instruction stream

1. Structural hazards

- What? – Resource conflict when HW cannot support all possible combination of instructions simultaneously/overlapped instructions
- How to eliminate? – ***Duplicating resources***

2. Data hazards

- What? – An instruction depends on result of a previous instruction in a way that is exposed by overlapping of instruction in pipeline; need to wait for previous instruction to complete read/write
- How do we eliminate? – ***Forwarding***

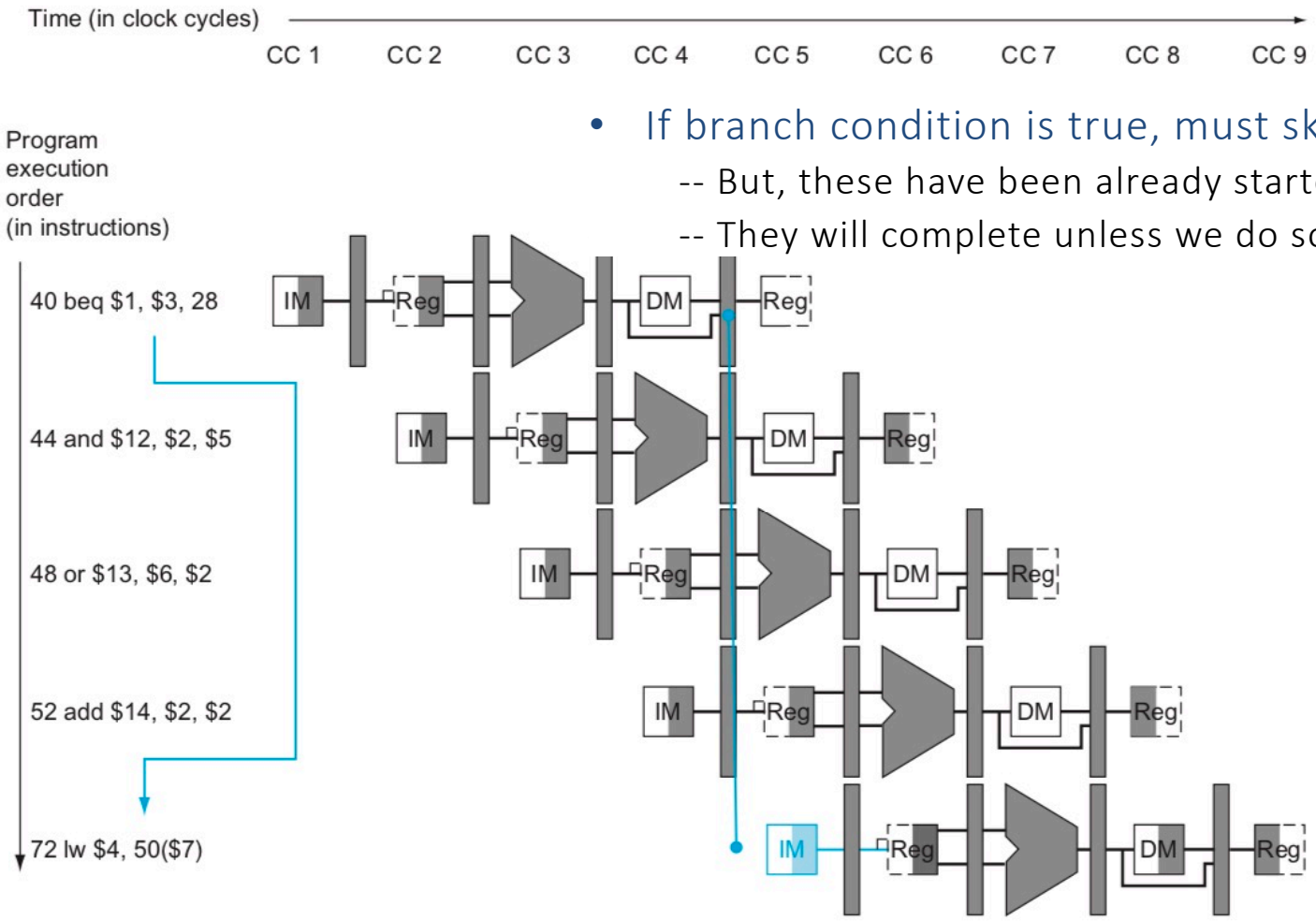
3. Control hazards – beq, bne, j, jr, jal

- What? – Pipeline of branches and other instructions that change PC (Program Counter); deciding on control action depends on previous instruction
- How do we eliminate? – ***Insert pipeline bubble***

III. Control hazards (Branch Hazard)

PIPELINE HAZARDS - CONTROL HAZARDS

- How branches impact pipelined instructions



- If branch condition is true, must skip 44, 48, 52
  - -- But, these have been already started down the pipeline
  - -- They will complete unless we do something about it

Pipeline Hazards - Control hazards

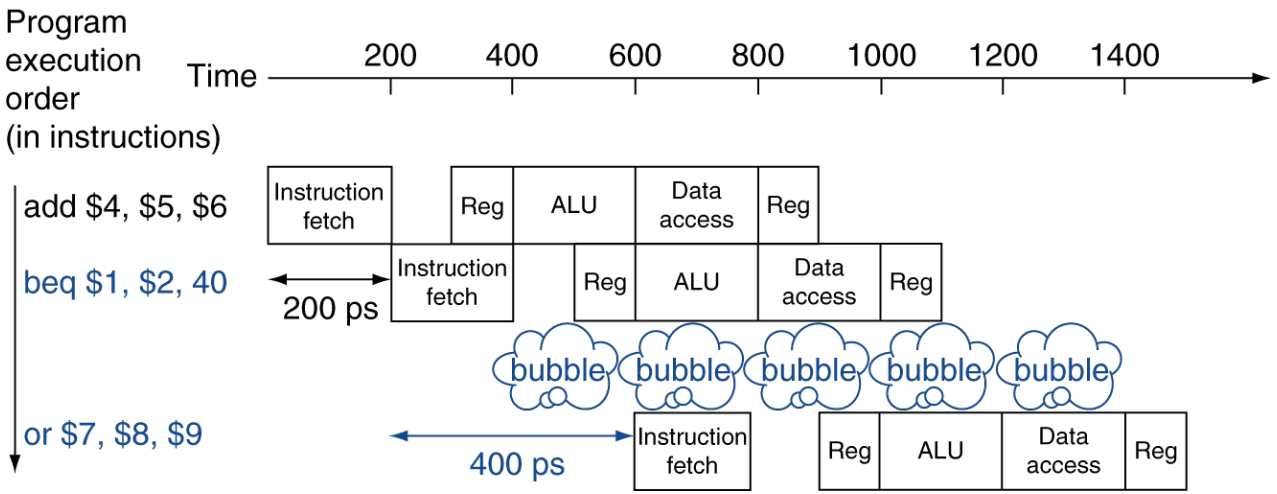
- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline cannot always fetch correct instructions

    --Still working on ID stage of branch

- Reduce Pipeline Branch Penalties
  - Freeze or flush pipeline
  - Predicted-not-taken (Predicted-untaken)
  - Treat every branch as taken
  - Delayed Branch

PIPELINE HAZARDS - CONTROL HAZARDS

- Dealing branch hazards: always **stall**
  - Wait until branch outcome is determined before fetching next instruction



  - If CPI = 1, 25% branch:

    -- Stall 1 cycle => new CPI = 1.25

    -- Stall 2 cycle => new CPI = 1.5

    -- Stall 3 cycle => new CPI = 1.75

PIPELINE HAZARDS - CONTROL HAZARDS

- Dealing branch hazards: always **stall**

- On average, branches are taken ½ the time
  -- If branch is not taken
    * Continue normal processing
  -- Else, if branch is taken..
    * Need to flush improper instruction from pipeline

- One approach:
  -- Always assume branch will NOT be taken
    * Cut overall time for branch processing in ½
  -- If prediction is incorrect, just flush the pipeline

Pipeline Hazards - Control hazards

- Dealing branch hazards: Branch Prediction (Static & Dynamic)

- Better (& more common): guess possible outcome
  1. Technique is called “branch predicting”; needs 2 parts:
     * “Predictor” to guess where/if instruction will branch; and to “where”
     * “Recovery Mechanism”: i.e., a way to fix the mistake
  2. Prior strategy:
     * “Predictor” : always guess branch never taken
     * “Recovery” : flush instructions if branch taken
  3. Alternative: accumulate info. In IF stage as to…
     * “Whether or not” for any particular PC value a branch was taken next
     * To “where” it is taken
     * “How” to update with information from later stages

Pipeline Hazards - Control hazards

- Dealing branch hazards: Branch Prediction (Static & Dynamic)

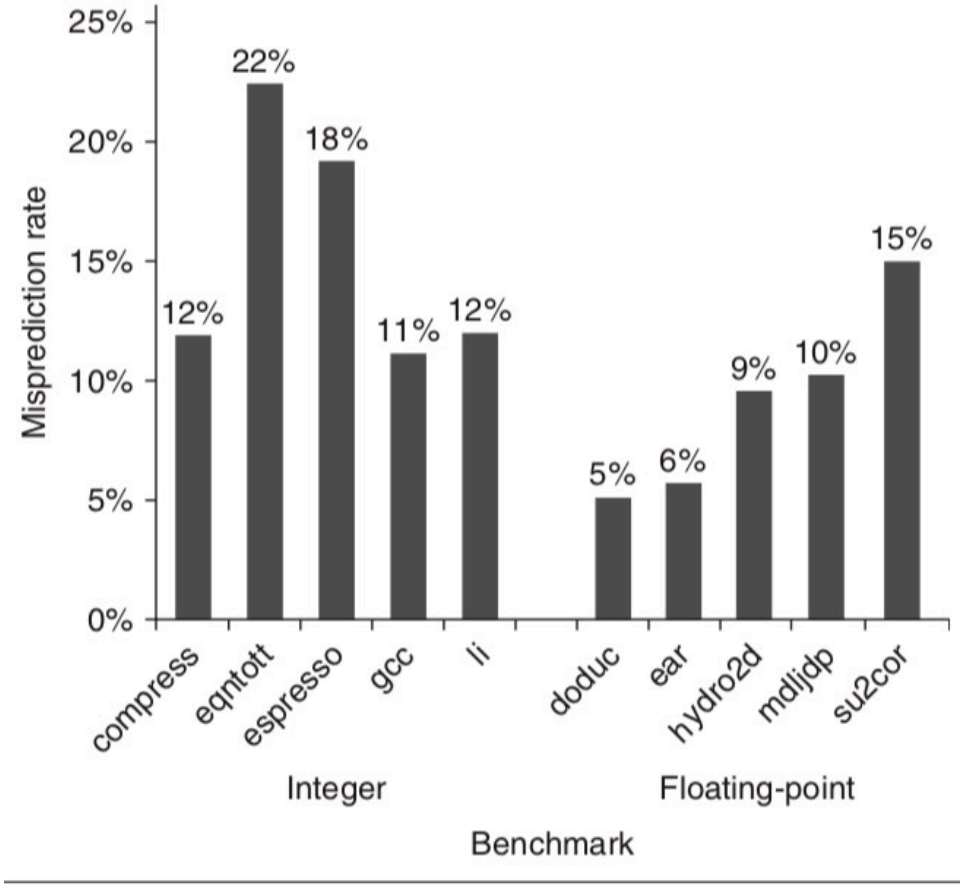
- Static Branch Prediction
  - -- Based on typical branch behavior
  - -- Example: loop and if-statement branches
    - * Predict backward branches taken
    - * Predict forward branches not taken

- Dynamic Branch Prediction
  - -- Hardware measures actual of each branch: e.g., record recent history of each branch
  - -- Assume future behavior will continue the trend
    - * When wrong, stall while re-fetching, and update the history

Reducing the Cost of Branches through Prediction

- Static Branch Prediction

  - An average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.
  - The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.



Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs

Pipeline Hazards - Control hazards

Dynamic Branch Prediction

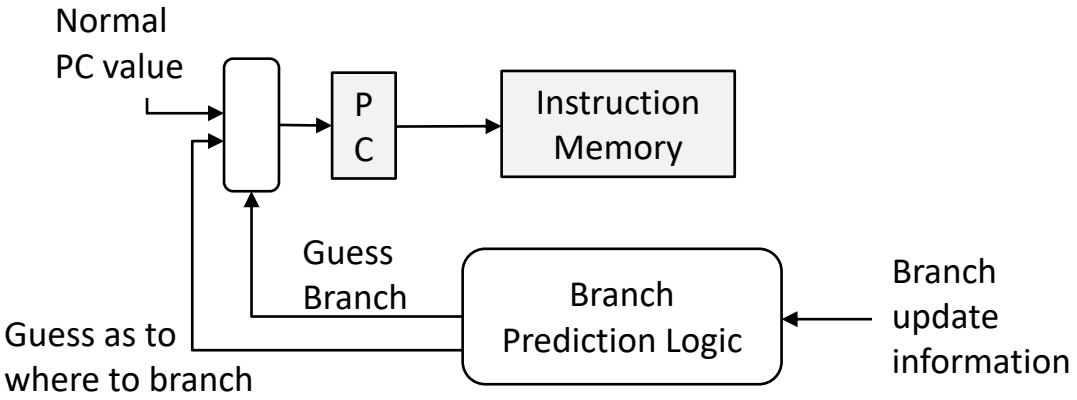
- **Why does prediction work?**
  - -- Underlying algorithm has regularities
  - -- Data that is being operated on has regularities
  - -- Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems

- **Is dynamic branch prediction better than static?**
  - -- Seems to be
  - -- There are a small number of important branches in programs having dynamic behaviors

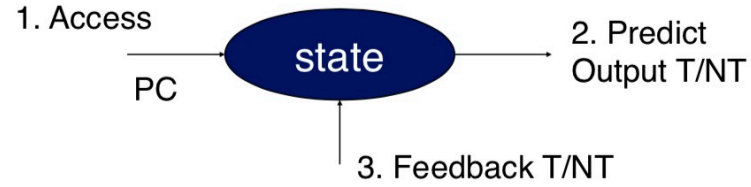
- **Two pieces of information needed for a branch**
  - -- Direction (whether the branch is taken or not), provided by Branch Predictor
  - -- Target (If it is taken, where it does), provided by Branch Target Buffer (BTB)

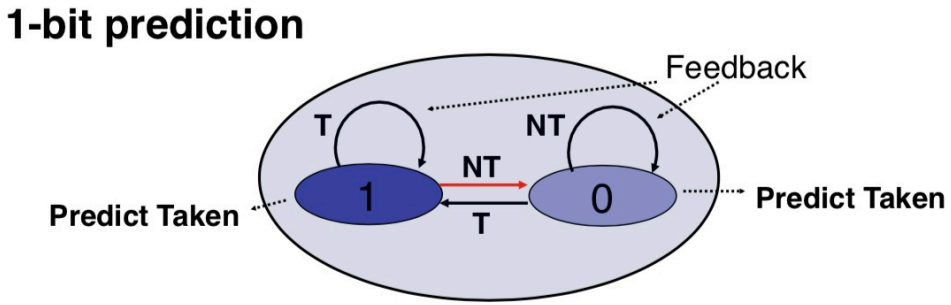
PIPELINE HAZARDS - CONTROL HAZARDS

- A branch Predictor



- General Form



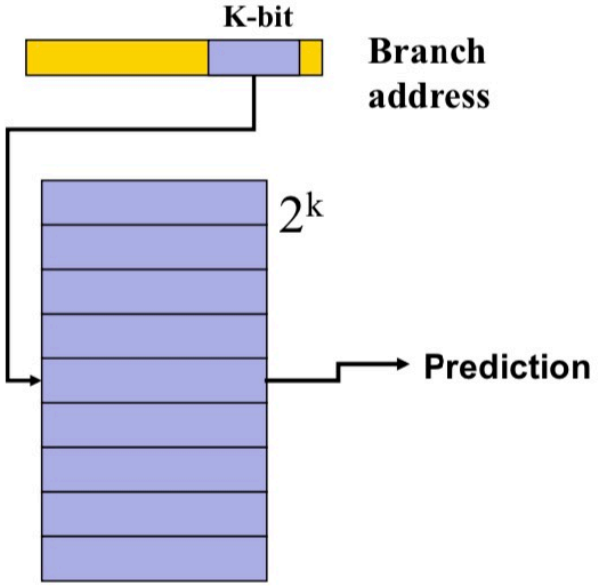


Pipeline Hazards - Control hazards

- Branch History Table (BHT) of 1-bit predictor

  BHT also called branch prediction buffer

  - Can use only one 1-bit predictor, but accuracy is low
  - BHT: use a table of simple predictors, indexed by bits from PC
  - More entries, more cost, but less conflicts, higher accuracy
  - BHT can contain complex predictors



- 1-bit weakness

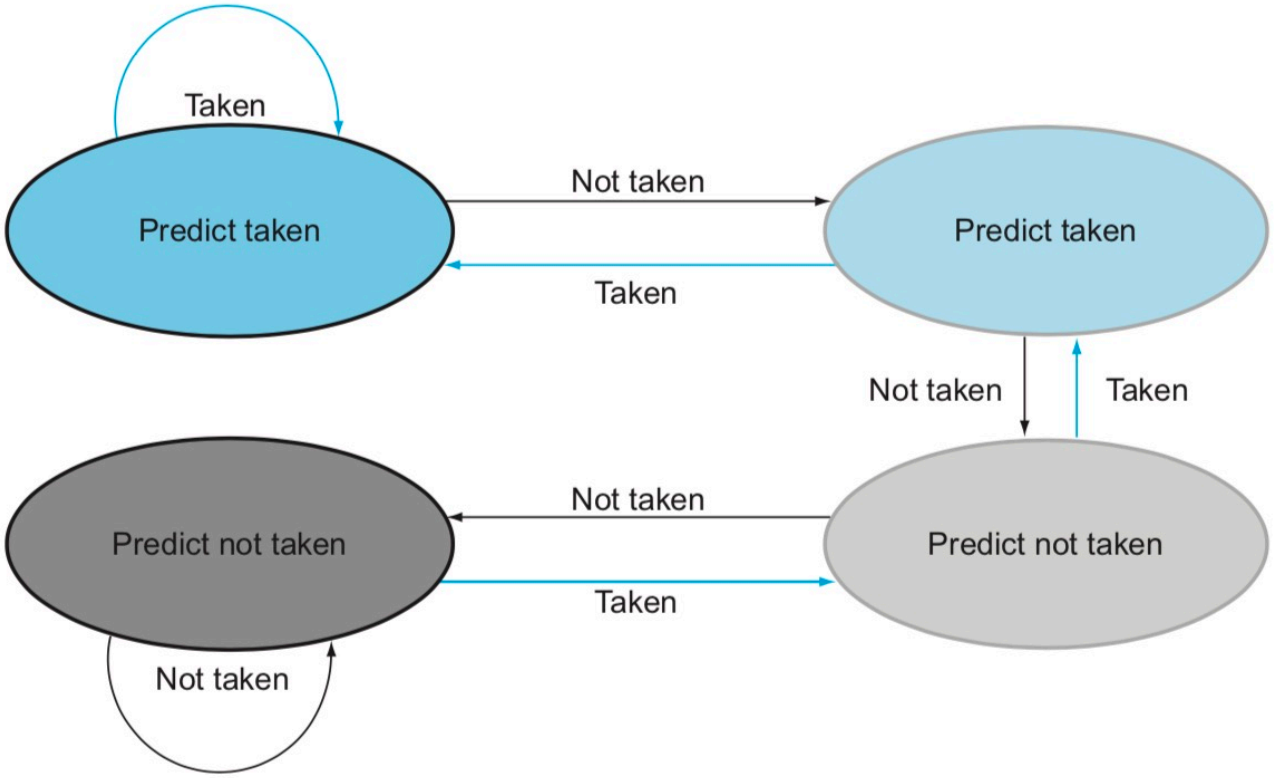
  - Example: in a loop, 1-bit BHT will cause 2 mispredictions
  - Consider a loop of 9 iterations before exist:

    - End of loop case, when it exits instead of looping as before
    - First time through loop on Next time through code, when it predicts Exit instead of looping
    - Only 80% accuracy even if loop 90% of time

```
for (…) {
  for (i=0; i<9; i++)
    a[i] = a[i] * 2.0
}
```

Reducing the Cost of Branches through Prediction

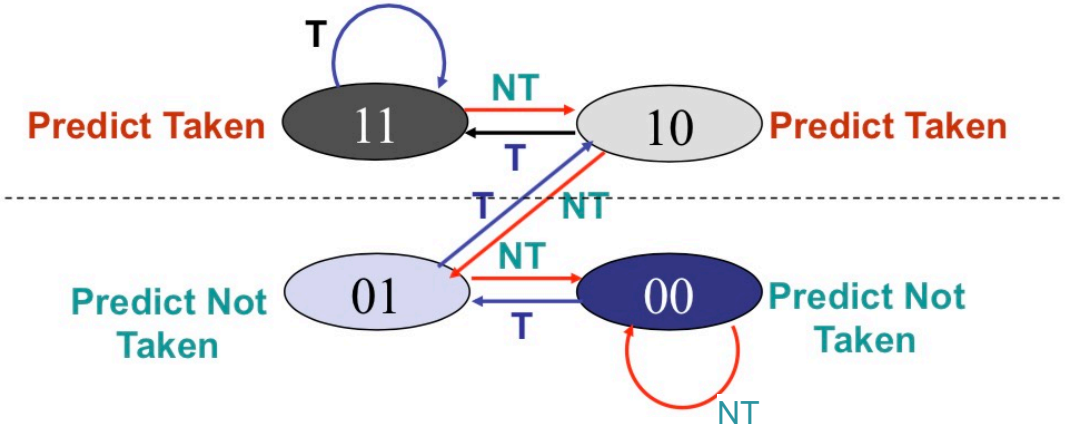
- Dynamic Branch Prediction
  - Branch-Prediction Buffers
  - Branch History Table



The states in a 2-bit prediction scheme.

PIPELINE HAZARDS - CONTROL HAZARDS

- 2-bit saturating counter
  - Solution: 2-bit scheme can change prediction only if getting misprediction Twice

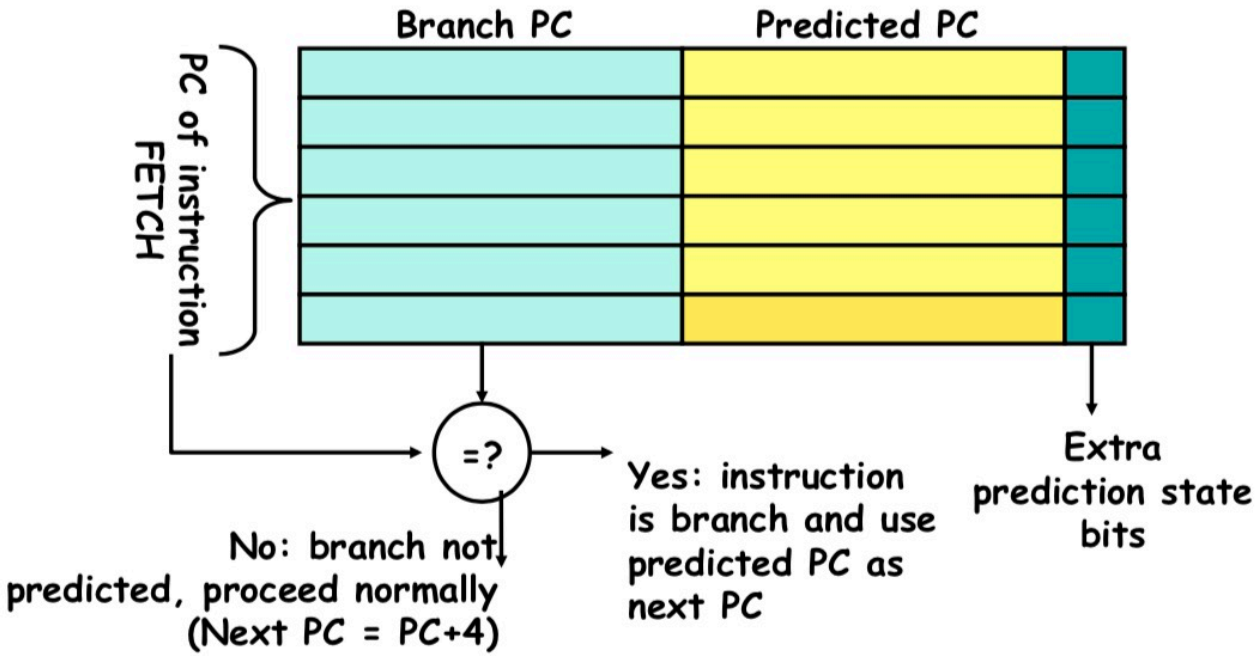


  - Blue: Stop, not taken
  - Gray: go, taken
  - Add *Hysteresis* to decision making process

PIPELINE HAZARDS - CONTROL HAZARDS

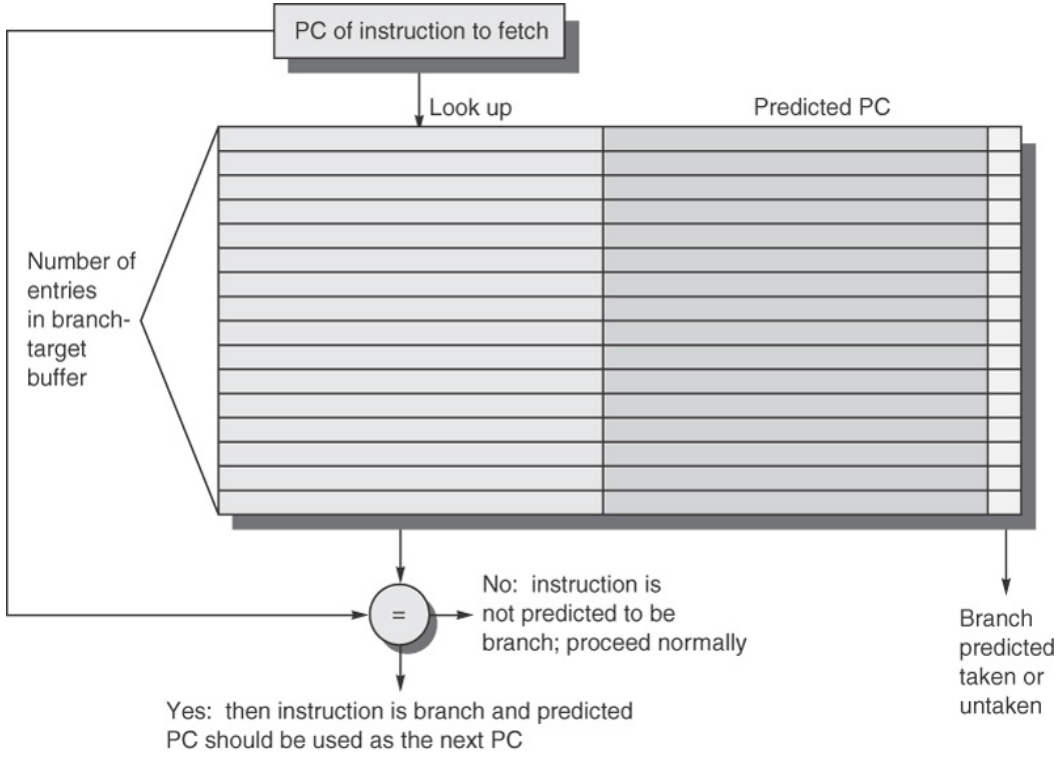
- Branch Target Buffer (BTB)
  - BTB: Address of branch index to get prediction AND branch address (if taken)

    -- Note must check for branch match now, since can't use wrong branch address
  - Example: BTB combined with BHT



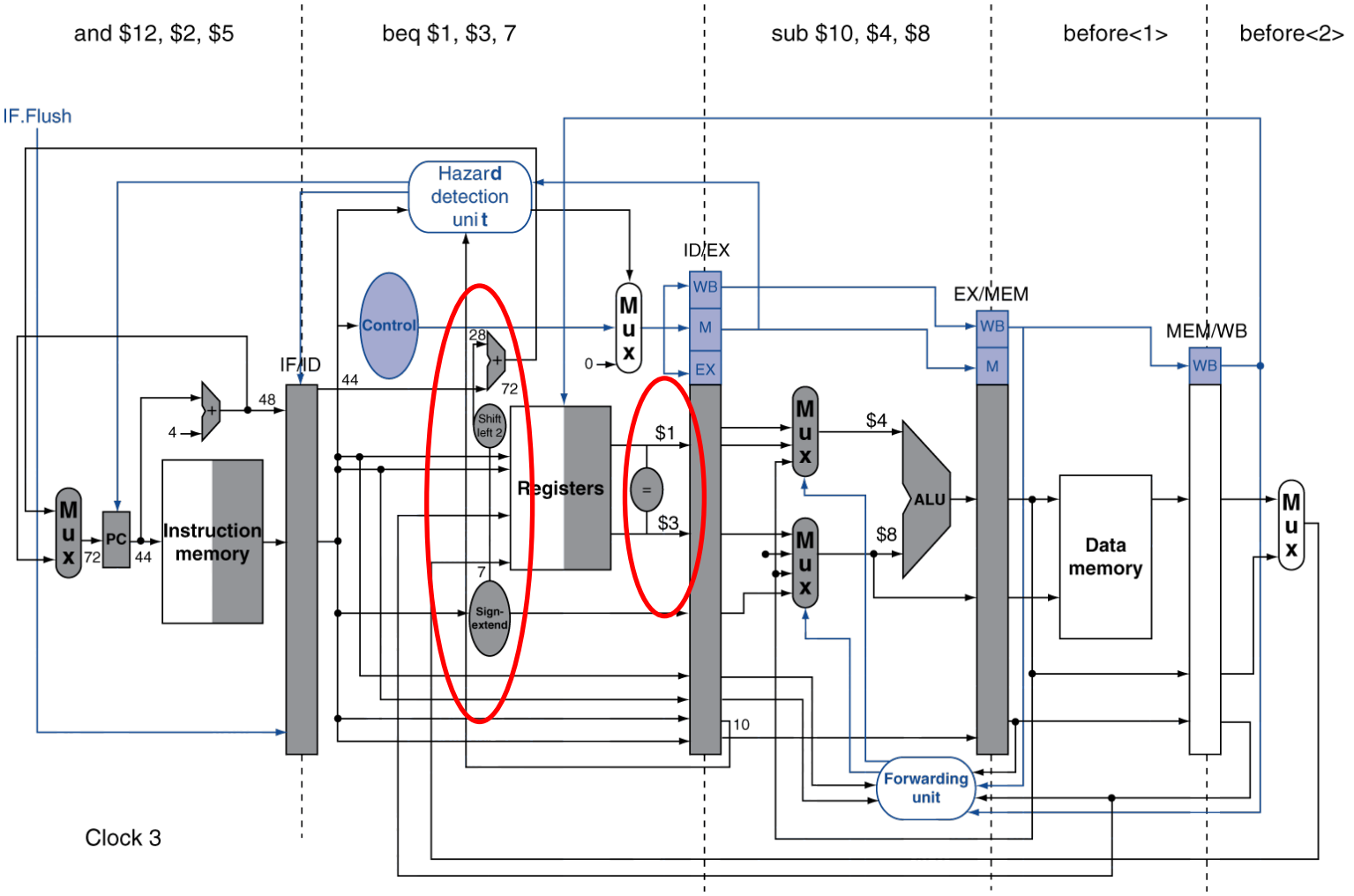
Pipeline Hazards - Control hazards

- Calculate the branch target
  - The BPB predicts when a branch is taken, but does not tell where its taken to!
  - Even with predictor, still need to calculate the target address
    - 1-cycle penalty for a taken branch
  - Branch target buffer
    - Cache of target addresses
    - Indexed by PC when instruction fetched (if hit and instruction is branch predicted taken, can fetch target immediately)



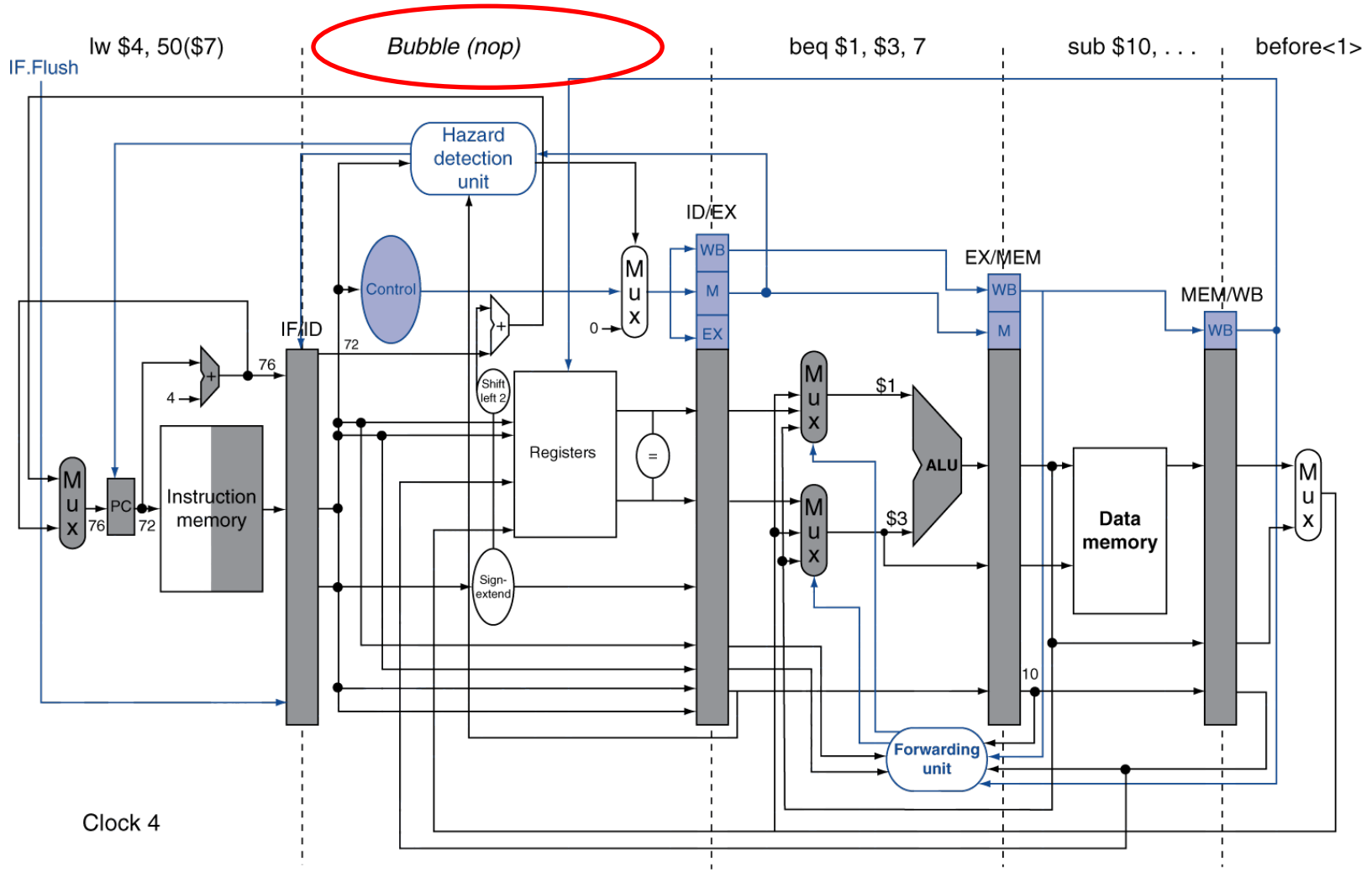
Pipeline Hazards - Control hazards

- Example: Branch Taken



PIPELINE HAZARDS - CONTROL HAZARDS

- Example: Branch Taken



PIPELINE SUMMARY: PERFORMANCE

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation`

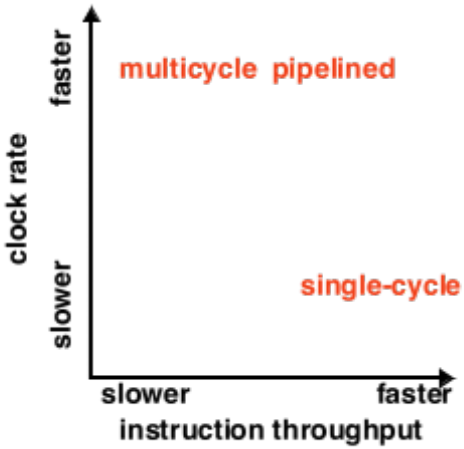
Throughput: instruction per clock cycle = 1/CPI

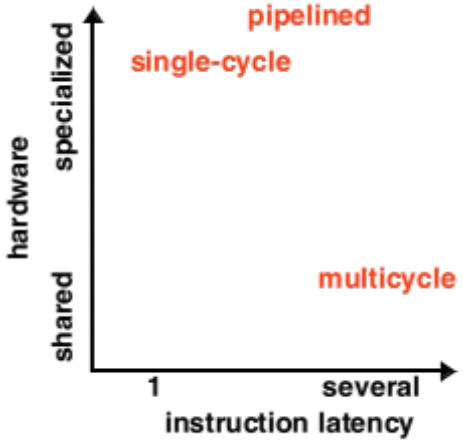
- Pipeline has fast throughput and fast clock rate

Latency: inherent execution time, in cycles

- High latency for pipelining causes problems

Increased time to resolve hazards





QUIZ

- Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
a = b + e; c = b + f;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from $t0:

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add   $t3, $t1,$t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
add   $t5, $t1,$t4
sw    $t5, 16($t0)
```

Find hazards in the preceding code segment and reorder instructions to avoid any pipeline stalls.

Answer

- Please reply your answers to TA before Friday (Mar 26, 2021)
- More detailed description will be better in your answers

Limitations of ILP for parallelable Processors

- Clock rate, logic complexity, and power
- Small issue rate and load/store buffers
- Longer latency

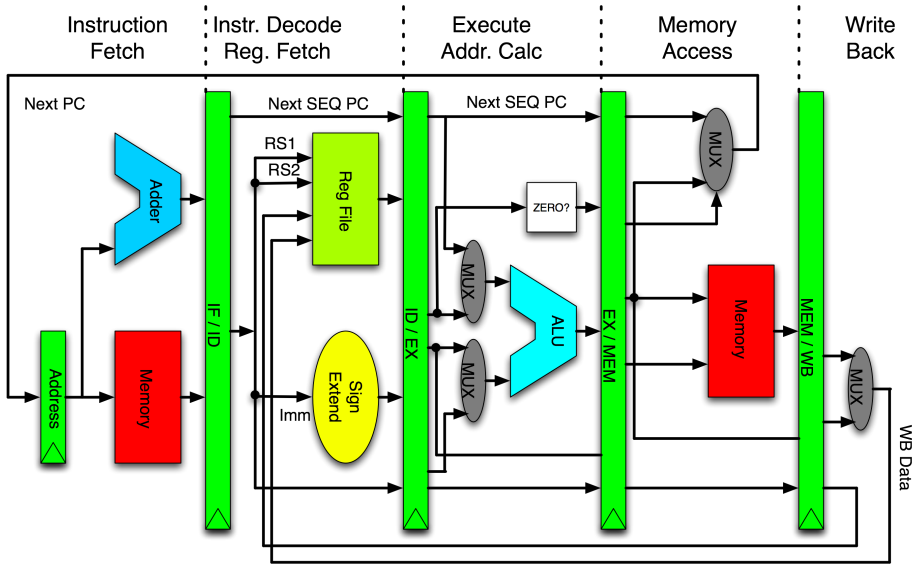
What Is Ahead??

What Is Ahead??

- ~2000, ILP in its peak
- ~2005, Multicore processor
- Then, superscalar processor
- Large number of cores
- Exploit DLP & TLP

What did we Learn So far

- Un-pipelining and Pipelining
  - ✓ Pipeline Stages
  - ✓ Performance comparison
- Instruction Level Parallelism (ILP)
  - ✓ Pipeline Processor
  - ✓ Superpipelined Processor
- Pipeline Hazards
  - ✓ Structural Harzard
  - ✓ Data Hazard
  - ✓ Control Hazard



Pipelining & Superscalar & Pipeline Hazard